DEVELOPMENT OF A RESPONSE PLANNER USING THE UCT ALGORITHM FOR CYBER DEFENSE

THESIS

Michael P. Knight, Captain, USAF

AFIT-ENG-13-M-28

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-13-M-28

DEVELOPMENT OF A RESPONSE PLANNER USING THE UCT ALGORITHM FOR CYBER DEFENSE

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Insitute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Michael P. Knight, B.S.E.E.

Captain, USAF

March 2013

AFIT-ENG-13-M-28

DEVELOPMENT OF A RESPONSE PLANNER USING THE UCT ALGORITHM FOR CYBER DEFENSE

Michael P. Knight, B.S.E.E.
Captain, USAF

Approved:

Maj Kennard R. Laviers, PhD (Chairman)	Date
Kenneth M. Hopkinson, PhD (Member)	Date
Lt Col Jeffrey D. Clark, PhD (Member)	Date

AFIT-ENG-13-M-28

Abstract

A need for a quick response to cyber attacks is a prevalent problem for computer network operators today. There is a small window to respond to a cyber attack when it occurs to prevent significant damage to a computer network. Automated response planners offer one solution to resolve this issue. This work presents Network Defense Planner System (NDPS), a planner dependent on the effectiveness of the detection of the cyber attack. This research first explores making classification of network attacks faster for real-time detection, the basic function Intrusion Detection System (IDS) provides. After identifying the type of attack, learning the rewards to use in the NDPS is the second important area of this research.

For NDPS to assemble the optimal plan, learning the rewards for resulting network states is critical and often depends on the preferences of the network operator. Using neural networks, the second area of this research demonstrates that capturing the preferences through samples is feasible. After training the neural network, a model can be created to obtain reward estimates. The research performed in these two areas complement the final portion of the research which is assembling the optimal plan through using the Upper Bounds on Confidence for Trees (UCT) algorithm.

NDPS is implemented using the UCT algorithm which allows for quick plan formulation by searching through predicted network states based on available network actions. UCT can effectively create a plan quickly and is guaranteed to provide the optimal plan, according to rewards used, if enough time is allotted. NDPS is tested against eight random attack scenarios. For each attack scenario, the plan is polled at specific time intervals to test how quickly the optimal plan can be formulated. Results demonstrate the feasibility of NDPS to be used in real world scenarios since the optimal plans for each attack type can be formulated in real-time allowing for a rapid system response.

This is dedicated to my wife for all her support that made this possible.

Acknowledgments

Thanks to my adviser, Maj Kennard Laviers, who provided the mentoring needed for this thesis. Also, I would like to thank my committee members, Dr. Kenneth Hopkinson and Lt Col Jeffrey Clark, who taught me concepts needed for this work and taking part in my defense committee.

Michael P. Knight

Table of Contents

Page

List of Figures

Figure Page

List of Tables

List of Symbols

Symbol	Definition
c	c parameter of UCT algorithm
n	n is count of instances
k	k is number of neighbors in ReliefF algorithm
rms	root mean square
H	Host reward value
R	Network reward value
w	Weight of Host
Q	Average Reward in UCT algorithm

List of Acronyms

Acronym	Definition
MDP	Markovian Decision Problems
UCT	Upper Bounds on Confidence for Trees
UCB1	Upper Bounds on Confidence
EM	Expectation Maximization
STRIPS	Stanford Research Institute Problem Solver
PDDL	Planning Domain Definition Language
AI	Artificial Intelligence
PSP	Plan-Space Planning
PoP	Partial-order Planning
CSP	Constraint Satisfaction Problems
SAT	Satisfiability Problem
IDS	Network Intrusion System
KDD	Knowledge Discovery and Data mining
WEKA	Waikato Environment for Knowledge Analysis
LR	Learn Rate
MSE	Mean Squared Error
NDPS	Network Defense Planner System
SFS	Sequential Forward Selection
GUI	Graphical User Interface
RTS	Real-Time Strategy

DEVELOPMENT OF A RESPONSE PLANNER USING THE UCT ALGORITHM FOR CYBER DEFENSE

I. Introduction

A significant problem with computer networks today is the inability to defend against cyber-attacks in real-time. Cyber defense was identified by President Bush and now President Obama as "one of the most serious economic and national security challenges we face as a nation, but one that we as a government, or as a country, are not adequately prepared to counter" [45]. As a result, the Comprehensive National Cybersecurity Initiative was issued by the White House. The initiative established three main goals: 1. establish front line defense for today's immediate threats, 2. defend against all threats, and 3. hardening the future cybersecurity environment. In the initiatives, an Network Intrusion System (IDS) system called EINSTEIN 2 was already underway that acted as a classifier detecting malicious traffic based on signatures. EINSTEIN 3, a new version of the IDS, is in development which is the next iteration with the ability to respond automatically before damage is done. The events described articulates the importance of the research in this thesis and highlights its relevance to an on-going cyber defense problem. This research investigates three areas which are crucial to developing an automated planner.

1.1 Three Important Areas for Automated Planning in the Cyber Domain

The goal of this research is to detect an attack and formulate a plan to counter the attack while it happens. While there are three required areas to achieve this goal, the primary area of focus of this thesis is using the Upper Bounds on Confidence for Trees (UCT) algorithm to assemble a response plan which is the function of the Network Defense

Planner System (NDPS). The first area of research needed to create an automated planner is to identify attack types in real-time by using a classifier. Defending attacks becomes more difficult when faced with overwhelming network features and samples to analyze. Classification is based on using the best network features to identify what type of attack is occurring. Once the type of attack is understood, the resulting network state from choosing an action can be estimated. This state is the first attribute an automated planner in the cyber domain needs to accomplish its task. After being able to estimate the network state, the next challenge is to identify what rewards to assign so the automated planner can find the optimal plan.

Learning the rewards for a given network state based on the network operator's preference is the second important area to building a cyber defense planner. Networks vary in composition and priority in functions performed over the network. This is why research is performed to find a way to capture what is important to a network operator. Samples can be created by querying the operator on a simple scale, say 1-10, on how bad a network state is according to their preference. Once the samples are created, a neural network can be use to output an estimated reward from what it learned by training on the collected samples. This method is investigated as a way to create a function to output rewards without having to assign a reward for every network state that could result given its current state, type of attack, and actions chosen. The last area is creating a real-time plan formulation that uses classification and reward calculations.

A method to search through the state-action space to build an automated planner in the cyber domain is the last area investigated. The NDPS uses the UCT algorithm to search through the state-action space of network states to assemble a plan which consists of a sequence of actions. The system performs simulations based on information network sensors would provide in a real world implementation. Cyber attack scenarios are randomly generated to verify if the performance of NDPS can operate in a real-time environment.

Results indicate it is feasible, but as the network size increases, other improvements to the NDPS are needed. These are the three areas are needed to create an automated planner in the cyber domain. The first area, real-time classification, requires selecting the best network features for accuracy and the best type of classifier for real-time performance.

1.2 Real-time Classification for Network Attacks

The classifier section of this paper focuses on the detection portion of the problem. Without being able to detect a cyber-attack, the process to create and carry out a plan to mitigate its effects cannot be developed. The plan should include information about the type of attack that is likely occurring in order to produce a set of actions or procedures to effectively combat it. By clustering and classifying network traffic, personnel or monitoring agents can more easily determine if the traffic is originating from an attack or otherwise normal activity. This classifier can be trained by a network to enable it to be able to identify normal traffic. Furthermore, if significant features are highlighted before the classifier is created, some complexity can be reduced allowing attacks to be detected more easily.

Clustering is the grouping of similar data items into clusters [17]. In this application, clustering is performed using the individual features of the network trace data to highlight the important/common features of attack strategies. The data-set used is the well-known Knowledge Discovery and Data mining (KDD) 99 which is network data produced from a simulated Air Force network experiencing different forms of network attacks. Waikato Environment for Knowledge Analysis (WEKA) is a software workbench design to support the application of machine learning technology to real world data sets [19]. WEKA is useful because of its versatility in allowing data to be presented in its own format, such as the KDD 99 data set and is designed to encompass most learning algorithms under a common interface. The algorithms used for this research are ReliefF, clustering by K-Means, and C4.5 also known as J48 in WEKA. Results show improvements in performance by clustering with a minor decrease in accuracy. Identifying the attack type accurately and quickly allows for better estimated network states which require reward learning to handle differences in networks.

1.3 Learning Rewards Using Neural Networks

Understanding network operators' preferences is key to implementing an effective network defense planning system. The planning system needs to be able to determine which network state is best in order to achieve the desired result. The reason for this is because the system will have to search through actions expected to improve the network state against attacks. As it searches through the action space, it evaluates how the network changes as a result of each action. Once it evaluates the network state after executing an action, it needs to know if that network state is preferred over another. For example, if the planner has a choice between two particular nodes being down, which node the planning system should choose should be based on the preferences of the network operator upfront. Computer network topologies vary significantly as well as the organization's perception of the importance of specific portions of the network, and so it is desirable to learn the preferences of the operator of a given network. This data can easily be gathered for a given network topology by presenting a rating scale to the network operator.

A multilayer neural network is the research area investigated to demonstrate the ability of a system to learn operator preferences [25]. Neural networks have shown promising results in learning applications [55]. One primary limitation this thesis explores is what needs to be done with the neural network when preferences are updated. This thesis explores the effectiveness and limits of using neural networks to learn reward values from network operators. This method shows promising results as long as the collected samples are reasonably precise as shown in Chapter IV. The last and primary focus research area is developing the NDPS using UCT.

1.4 NDPS, an UCT Automated Planner

Analyzing the performance of the UCT algorithm which is what NDPS uses to create a plan is the primary focus of this thesis. UCT is a efficient search through Markovian Decision Problems (MDP) environment and is well suited to the cyber domain with its real-time requirements for defense [31]. There have been other attempts of using automated planners for cyber defense using graph plan algorithms [6]. This research shows the performance of the UCT algorithm under two different network configurations. This thesis demonstrates that the novel use of UCT for cyber defense is feasible and could be very beneficial.

An automated response system can be created to counter cyber attacks to minimize damage to computer networks. Human-computer interaction stands to benefit from this planner that would allow the network operator to quickly see the best course of action. Once the best course of action is determined, operators can manually execute the plan or perform similar actions they deem more appropriate. A reasonable goal is an automated planner that can perform significantly faster than a network operator and is the research target of this thesis.

II. Literature Search

2.1 Introduction

Automated planners require many areas of research to be integrated for the many domains that they are implemented. First, the planner has to gather information from the domain it operates. In the cyber defense domain, abstracting the information from the network environment is a difficult task. There are vasts amount of data transmitted quickly through large networks. As a result, research in detecting attacks has mostly been by performing classification of attack signatures. Numerous works propose unsupervised machine learning methods to classify network information using neural networks, K-Means, and flow learning [10][13][56]. Another work uses Expectation Maximization (EM) clustering to capture network features [40]. To facilitate processing large numbers of network features, feature selection by ReliefF is used in [32]. These works provide the necessary insight to abstracting the data of the best network features to allow faster classification using two well known classifiers, C4.5 and K-Means [47][39]. Another requirement of planners is the ability to capture the right information from the domain in order to make reward calculations.

Neural networks offer one approach to perform unsupervised machine learning [55]. Since its inception, many variations to the method have been made. Neural networks have been expanded into many layers that can perform back propagation calculations to train on samples [54][49]. Additional modifications into the activation functions which gives the output based on calculations inside the neural network have been researched[29][41]. These modifications to neural networks have enabled the components necessary learn reward values from network operators. Generally, this information is obtained by expert knowledge in the field. However, extending it to other networks requires learning models for portions of the network that are important to the network operator. By gathering samples, calculations can be made to estimate rewards from the trained neural network. Having the relevant rewards is critical when searching to assemble a plan.

The last key area of automated planners is the method used to search. Using abstracted domain information, many methods for search have been created. Automated planners were first introduced in 1971 in a system called Stanford Research Institute Problem Solver (STRIPS) [16]. These early planners are known as classical planning which are state, plan, or a mixture based [21][3][53]. Classical planners are deterministic and fully observable. Planners often require learning a strict programming language to input domain information and consequently the time to create a plan can grow exponentially with respect to the information they use. Another type of planner which is stochastically based is called Markovian Decision Problem (MDP) [4][27]. This type of planner can work in a stochastic domain. The state information for this kind of planner assumes all historical information from previous states. UCT is a subset of MDP planners.

UCT operates on MDP assumptions and efficiently searches through the state-action space. First introduced by applying the Upper Bounds on Confidence (UCB1) algorithm to trees where each node represented a state [1][31]. In these works, the algorithm was shown to operate efficiently and is guaranteed to find the optimal set of actions given enough time. It retains the most optimal choice during the search of the state-action space. Many applications have implemented this algorithm in gaming domain. One notable accomplishment for the algorithm was used in the game Go to defeat a master level Go player [20]. Go is a considerable challenging domain as a result of its large combinatorial state space, in fact larger than Chess. Real-Time Strategy (RTS) game domains have also used UCT [2]. The RTS game Wargus uses UCT to generate plans that the work shows is superior to baselines and human players. UCT has even been implemented in opponent modeling for the sport of football [34]. In this work, UCT is used repair plans made in simulated football games in real-time. These works show the advantages of using UCT and highlight its suitability to the cyber defense domain.

2.2 Classifying Network Attacks

There has been a great deal of research in the area of learning feature representations from unlabeled data sets for high-level tasks such as classification. Much of this research has shown great progress on benchmark data sets like NORB and CIFAR, object recognition data sets, by making use of complex unsupervised learning algorithms [10]. The WEKA system is traditionally used with agricultural data sets and these data sets tend to be larger and of lesser quality than those in machine learning data sets [18]. Applying the WEKA toolkit to machine learning data sets allows us to answer questions like "Do these changes in certain features indicate a cyber attack?"' as well as gain some insight into how to apply learning algorithms to existing real world data sets. Previously, the classification of network traffic was performed through the use of port-based or payload-based analysis. This has become increasingly more difficult as peer-to-peer networks (P2P) adapt to using dynamic port numbers, masquerading techniques, and encryption to avoid detection [13]. To combat these cyber-attack adaptations, an alternative approach for classifying network traffic is introduced by exploiting and extracting common or distinct attack strategy characteristics. Authors of another related paper [40] describe their efforts using the Expectation Maximization (EM) algorithm [12] to cluster network flows into different application types with a fixed set of attributes (features). The EM algorithm separates the network traffic into a few basic classes but the accuracy and quality of the clustering is limited as a result of its fixed attributes. Another technique uses the Sequential Forward Selection (SFS) algorithm [56] to find the best feature set to avoid an expensive, exhaustive search. Some other data sets, similar to KDD 99, used with the SFS feature selection algorithm include the Auckland-VI, NZIXII and Leipzig-II traces [57]. This thesis uses the popular and well-known KDD 99 data set for experimentation and evaluation purposes. A modification of the Relief algorithm [32] is used for feature selection and clustering is performed via the K-Means algorithm [39]. Additionally, the C4.5 algorithm [47] is used to build a decision tree based on the feature selection results. Identifying the type of network attack is one piece of abstracted information that is needed from the cyber domain, the other is to be able to learn the rewards for a given network. This thesis explores the use of neural networks to estimate reward values.

2.3 Neural Network Reward Learning

Neural networks are typically used as a method for unsupervised machine learning. They were introduced as far back as 1960's [55]. Since then many modifications have been proposed to improve the learning of neural networks. Improving the activation function that is used in training is one such modification [29][41]. Another is by creating additional layers in the neural network that offers non-linear learning [54][49]. These expansions have enabled several more research applications to be performed. In the cyber domain, neural networks have been used for many classification applications. One application is to block certain web content by training neural networks [35]. The idea behind this application is to train the neural network based on samples gathered from web pages and extracting features from those samples. Another similar type of application using neural networks is to create an Intrusion Detection System (IDS) by training a neural network [50]. Instead of using other classification methods, this work uses neural networks to classify network attacks by training a neural network using the KDD99 data set. In addition to classification, neural networks can also be used for learning rewards. An example research application that uses neural networks to learn rewards is robot exploration [38]. Even in simple tasks, estimating rewards is not an easy tasks for robots due to the complexity of maintaining state historical information. Neural networks are used to train and update reward information iteratively. Estimating the reward information from network operators is a novel research application. As a result, this research is performed in this thesis to see its effectiveness. Once the reward information is determined, the next logical step is to develop a search technique to find the sequence of actions that will lead to the highest possible reward.

2.4 Automated Planning

To find the best sequence of actions, this work includes a planning algorithm. In general, planners combine two major areas of Artificial Intelligence (AI) which are search and logic [48]. Planners have the difficult task of managing an exponential number of propositions to create a plan composed of actions. Along with this, most planners have to be configured and tailored to their environment to operate properly [22].

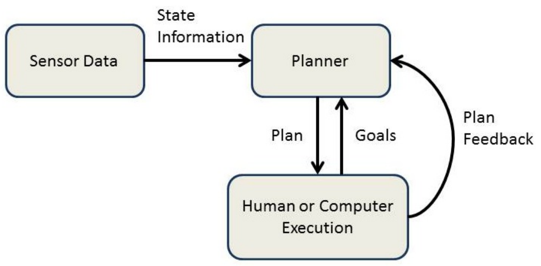


Figure 2.1: This is a general diagram of how the planner interacts with environment.

Figure 2.1 shows a layout of how a planner would interact with other components of a typical system and work to formulate plans which are composed of actions. The planner uses sensor data to formulate an abstracted state for it to analyze. There are multiple aspects of planning which have led to different kinds of automated planners.

2.5 Classical Planning

Classical planners use a restricted state-transition system [22]. This is a deterministic, static, finite, and fully observable system. These planners start at an initial state and formulate a list of actions to reach a goal state. The state-transition system has to manage a large combinatorial search space. In addition, this kind of planning has the problem of defining all enumerations of states and actions which can be difficult for some tasks. These types of planners are hard to use for general purposes as they require domain specific information to operate.

Generally, the classical planners focus on building a solver for general models and are domain-independent. The most well-known classical planners include Stanford Research Institute Problem Solver (STRIPS) and Planning Domain Definition Language (PDDL). STRIPS was the first major planning system developed in 1971 by Fikes and Nelson. PDDL [22] came around much later and has been used as the standard for International Planning Competition ever since. PDDL requires the initial state, available actions, goal requirements, and results of actions to search. PDDL is still around as PDDL3.1 and there have been many variants from it. These are some of the early classical planners and were examples of state-based planning.

2.5.1 State-Based Planning.

State-based planning is a subset of the classical planners which uses AI search techniques to formulate the best course of actions by performing a search until a set of actions is found that reaches a goal state. Each node that it searches represents a state of the world and each line from that node represents a transition to another state. Figure 2.2 provides an example of a typical planning problem with stacking blocks to demonstrate a classical state-based planner. Trying to move a block from one location to another requires managing the actions that lead to the needed states so that the block to reach its destination. This example is a simple environment but for larger environments this causes the search problem to grow exponentially. These problems demonstrate some of the concepts of planning such as preconditions, actions, and states that classical planners use.

Some state-based planners are incomplete because of their search techniques. Techniques such as forward searching use a depth-first search like approach that can lead to incomplete searches because of infinite paths. One of the first planners, STRIPS,

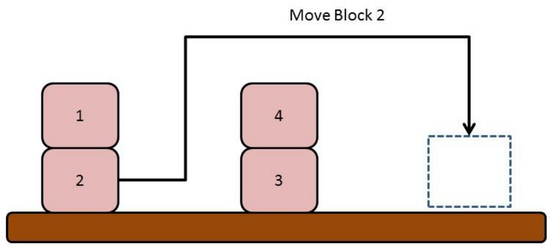


Figure 2.2: Sample of simple planning problem involving stacking blocks.

used a state-based planning approach. This style of planning seems intuitive, however, overcoming the vast search space of the problem and making it efficient remains a complex task, but there has been advancements that help guide the searching of these state-based planners to make them faster.

Some state-based planners that are non-interleaved have an issue called Sussmans Anomaly. This anomaly is where the planner makes subgoals to reach the main goal but has to break the subgoals to reach the main goal and vice versa depending if they swapped them around. Planners can account for this problem now but demonstrate the complexity inherit in state-based planning.

2.5.2 Plan-space Planning.

Another family of planners are the so called plan-space planners. Plan-space planners seek to refine partial plans represented by the nodes. Plan transformations are represented by the lines between the nodes. These planners tend to have smaller search spaces since they ignore the states and focus on ordering the plan's actions to reach the goal state. However, they can search infinitely if the algorithm does not include a mechanism to detect cycles while searching through the plan-space.

Plan-space planning uses a relaxed version of goal finding by not constraining the plan order and searching through a set of partial plans to develop a plan that meets the goal conditions. Plan-Space Planning (PSP) is a search procedure that uses this. Partial-

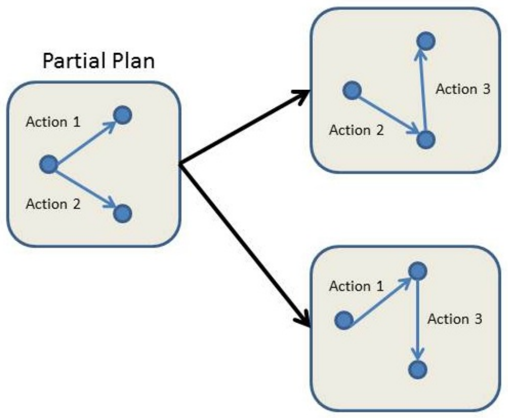


Figure 2.3: Plan-space searching by transforming partial plans.

order Planning (PoP) is a modification to PSP and processes flaws differently in a partial plan. PoP processes flaws with respect to its subgoals and PSP heuristically selects flaws to be refined. These two are the main algorithms that perform planning through plan-space which searches through partial plans. Typically, plan-space planners do a backward search from the goal to the initial state to formulate a plan. It does not make any commitments while searching so if it needs to use an action but that action has a precondition that needs to be satisfied, then it goes to search for an action or plan to meet that precondition.

2.5.3 State-Based vs. Plan-Space Planning.

There are several differences in the two search spaces. State-based is finite while plan-space can be infinite. State-based planning has to have explicit states where everything in the state has to be defined. This is not true for plan-space planning. Plan-space planners were previously faster performing since their search space tended to be smaller but over the years have been outperformed by state-based planners [3, 53]. Partial plans have the drawback of losing the explicit state information since it does not account for it. This makes them harder to control without that information. With the advancements of state-based planners, plan-space planners are not as efficient computationally anymore.

2.5.4 Hybrid Approach (State/Plan-Space).

There is a faster hybrid approach of the two types of planners called planning-graph in which the algorithm searches through a planning-graph space. The Graphplan planner [5] brought a new level of planning performance to the research community. Planning-graph uses an iterative deepening approach to search through the space [3]. This structure accesses which propositions are reachable from the set of actions it can take.

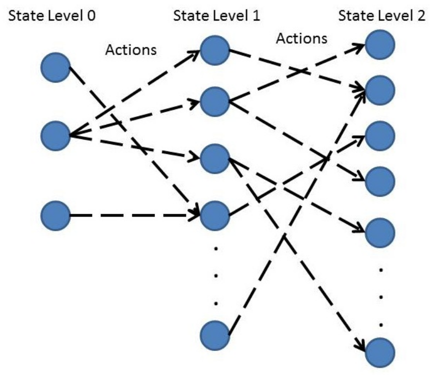


Figure 2.4: Picture of the layers of Graphplan Algorithm.

The graph is layered with arcs only going between layers. These layers consist of state nodes which are connected by precondition lines to the actions the planner has to available to it. After reaching a certain depth, the solution is found by backward searching through these layers. It has been shown that Graphplan is complete and performs polynomial time. There have been enhancements to the memory space required and performance increases for the searching. There is still a level of complexity involved similar to the other classical planners. Since these are domain-independent, the domain information has to be translated properly into the algorithm. Another approach to planning is planning as a satisfiability problem.

2.5.5 SAT/CSP Planning.

The technique of planning as a Satisfiability Problem (SAT) planning is another type of a classical planner. Satisfiability problems are generally a set of clauses that are resolved by assigning truth values to each literal in the clauses. Figure 2.5 below shows an example of a SAT clause to give an idea of how it formulates its goal criteria and what it needs to resolve to find a set of truths to satisfy this condition.

$$(P \vee Q) \wedge (R \vee \neg S) \wedge (\neg Q \vee S \vee R)$$

Figure 2.5: Example of SAT clause.

This has been shown to be NP-complete. This is the same for Constraint Satisfaction Problems (CSP) as well. CSP planning is similar to SAT planning. There are differences in encoding the problems to CSP or SAT but otherwise are similar. SAT planning is one of the more current used classical planners. It translates planning problems into satisfiability problems. After propositionalizing the actions and goals and defining the initial state, the information can be given to SAT Plan algorithm to solve. The complexity involves encoding the planning problem into a satisfiability problem. The procedures used in this technique are complete and sound.

2.6 Markovian Decision Problems (MDP) Planners

A type of planner that deals with uncertainty is MDP planners and as the name implies, uses the MDP to deal with nondeterministic states. The domain the planners operate in is a stochastic system. MDP planners are seen as optimizing to create the best plan. It uses a utility function to maximize what it calculates to be optimal. These planners can be both implemented under the assumption of full observability and partial observability. The assumption with MDP is that in a given state all of its history is captured in the values of that state. The next state would then rely solely on that previous state for all the historical information to calculate its values. Figure 2.6 shows the general representation of how the MDP are organized. They have states and transition to other states by performing an action with a given probability to get them there. There are two main methods to optimize MDPs [22].

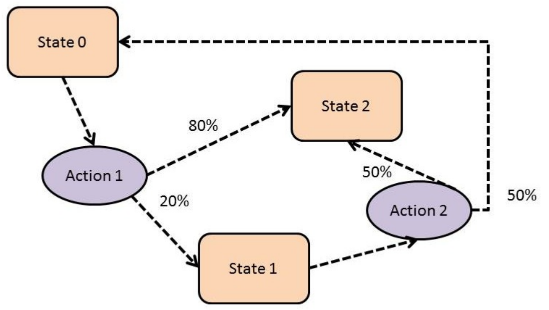


Figure 2.6: Example of states and actions with probabilities associated with them.

Value and policy iteration are two ways to optimize MDPs. Value iteration finds the best value for a given policy by doing iterations until the values converge. The policy is the method used to determine the next action to take at a given state. For example, a policy for moving around in a grid world might be; randomly choose a direction or go in the direction with the highest value. The other method is to optimize the policy through policy iteration. Policy iteration is performed by first selecting a policy and evaluating it to find the best values. After this, the policy is improved based on using the selected values and then the process is repeated. The process loops until the policy becomes stable and consistent between iterations. Another method similar to policy iteration is the UCT algorithm which is used on MDP's. Unlike the two previous methods, UCT is flexible in that it does not have a convergence requirement and can provide a best course of action given a limited amount of time. UCT is an efficient search that always provides the best solution for the parts of the search tree it has explored.

2.7 UCT Related Work

UCT was introduced in 2006 to improve "rollout-based" Monte-Carlo planning by applying the bandit algorithm, UCB1 [31]. The bandit algorithm was created for the multiarmed bandit scenario where a bandit pulls the handle of several slot machines to get the maximum payout [1]. The key question is whether to exploit a good reward path or explore other paths for potentially better rewards that would come down another path. UCT expanded on the UCB1 algorithm by applying UCB1 to a search tree. This tree analyzes the action-state space. These algorithms are based on a Monte-Carlo search and improve the performance by using statistically guided search methods instead of random selection of actions. It was demonstrated in [31] that the algorithm improves efficiency significantly while maintaining optimality. The work shows that there is a constant value, c, used in the algorithm that is domain dependent. This c value guides the search to various degrees of exploitation vs. exploration to gain the best average rewards. From UCB1 [31], a value of $c = \sqrt{2}$ is introduced as getting a balance of exploiting and exploring.

UCT algorithm is used in various types of games. One application of using the UCT algorithm was done in real-time strategy games [2]. In this work, UCT is used as an automated planner in the RTS game Wargus. The results show the algorithm was the top performer compared to several other baselines and human players. It was also the only planner to find the winning strategies for all scenarios with the exception of the human player. UCT was able to optimize several different winning criteria which makes RTS games complex domain. Go [20] is another game in which the UCT algorithm is used. The program is called "Mogo" and was able beat a human opponent at the master level. UCT algorithm was demonstrated to efficiently find optimal results in several complex domains.

UCT algorithm uses Monte-Carlo sampling to generate sparse trees until the children of that given node have all been visited once. Afterward, the algorithm chooses an area to explore based on a calculation using the expected reward, Q, which is calculated by Equation (2.1) each time a sample is taken. The variable s is defined as state, a is the action, and R stands for the resulting reward value for the last state visited on a given iteration.

$$Q(s,a) = Q(s,a) + \frac{1}{n(s,a)}[R - Q(s,a)] \tag{2.1}$$

The policy, π, uses an upper confidence bound, Q^{+}, which is calculated using Equation (2.2). Equation (2.3) defines how π is chosen by maximizing Q^{+} value to select the next area to explore.

$$Q^{+}(s,a) = Q(s,a) + c \times \sqrt{\frac{log\ n(s)}{n(s,a)}} \tag{2.2}$$

$$\pi = argmax_{a}\ Q^{+}(s,a) \tag{2.3}$$

The algorithm does not have many parameters to initialize but carefully choosing c can have a significant impact on performance and is demonstrate in several UCT research applications [20][2]. Other parameters influencing the performance is the cardinality of the action set for each state and how deep in the tree the algorithm searches. These parameters influence the time and space performance of UCT algorithm implementation and must be considered. UCT with its ability to provide the best answer given a restricted time window lends well to quick results a much needed trait in the cyber defense domain. Other automated planning algorithm used in the cyber domain have also been researched.

2.8 Cyber Defense Planning Related Work

There are not many automated planner algorithms implemented in the cyber domain with some notable exceptions. One automated planning algorithm that was implemented was PDDL [6]. This classical planner requires expert knowledge of the PDDL programming language to input domain information. The system builds plans based on preconditions that are manually programmed. The real-time performance needed to generate a plan is exceeded once the number of objects increases. With approximately 200 objects generated in the domain, 200,000 actions are generated and the report time for the plan can be near 33 seconds [6]. Cyber attack types have different requirements in terms of response time but 30 seconds of time to formulate a plan could be potentially to long to respond to some scenarios. In addition, the load time of the domain can be considerably long depending on the number of objects placed in the domain. After a certain number of objects, exponential growth starts to happen. UCT suffers from exponential grow of the state tree as well, however, the growth can be controlled by searching at controlled depths to formulate the plan. Future foresight of actions is lost the less deep in the tree the algorithm goes. The algorithm gives flexibility in terms of providing the best plan it can find given the time allowed. Flexibility in terms of run-time is a needed attribute when creating a cyber defense plan. PDDL is not really flexible in terms of building a plan and incorporating new domain information. There are more user friendly tools to insert domain knowledge, but there is no potential to explore new actions and learn rewards as it currently exists.

2.9 Best for Cyber Domain

MDP planners offer promising routes to explore. MDP planning is better suited for the nondeterministic domains and may be easier to deal with if the planner cannot get all the information necessary and has to make reasonable estimates. There would likely have to be a learning component to MDP to acquire good policies and determine the best value of taking certain actions. However, this adds more complexity to setting up the planner as there is likely to be more information to consider and process. This is because there are many unknowns initially until the planner learn the best policy. Even after learning a policy, the environment can change to alter the values to make it that the policy is no longer best. The policy has to be learned in each domain of the planner. Perhaps with enough learning, the policy will be able to perform well enough even if the domain changes slightly. UCT is a search algorithm that exists in the MDP domain. It is one of the more efficient methods to search a MDP state-action tree. It uses the policy of finding the best average reward and using an upper confidence bounds to explore the state-action tree.

UCT offers the advantage of flexibility which is an important component in building an automated planner in the cyber domain. Network attacks vary in terms of execution time so having an automated planner that is able to output a plan anytime is critical. Nearly all the other planners described previously have the problem of having to assemble plans by going through most of the combinatorial space of objects in the planning domain. As a result, search time is significantly longer since it does not terminate until preconditions to actions and states are met. UCT does not have this restriction since it assumes a Markovian state space. All the information about a state is assumed to capture the history of that state. As a result, the state estimator can be a challenge as it must be able to figure out the possible states and maintain historical information. However, only information needed from the domain can be captured, and if understood well enough, should not hinder it. Even if the information could not be gathered from expert knowledge, it is possible to learn probabilities of the estimated states occurring through training. Another benefit of using UCT as the planning algorithm is that the search is efficient while searching for the optimal plan. This is due to using upper confidence bounds on expected rewards discovered while searching. Using one parameter gives control over how much exploration is needed. A desirable parameter is necessary in the cyber domain due the need to find the plan quickly.

The efficiency and flexibility make UCT an ideal choice to implement an automated planner for cyber defense.

III. Real-time Classification for Network Attacks

Even after an IDS has identified a cyber-attack, network administrators are still faced with the difficult challenge of assessing network health and status in order to appropriately take action to mitigate damage caused by such an attack due to the large amount of data available from the network components. This chapter explores the use of auto-clustering to abstract network meta-data to form high-level units of information that are more comprehensible for a network administrator, or an AI Agent to understand and act on. An empirical analysis is performed to evaluate the approach using the NSL-KDD99 data set for both abstraction of network log data and attack family classification. By auto-clustering, the classification speed is significantly increased without greatly increasing the error.

3.1 Method

The approach for real-time classification is to use a familiar data set, KDD99, to abstract the data into clusters and classify based on the abstracted information. There is an inherent loss of accuracy by using this method. However, the minor decrease is for faster classification which is a crucial element for the cyber domain. Abstracting the data is done after performing feature selection using ReliefF to reduce the 41 features of the data set. Once features are selected, the Expectation Maximization (EM) algorithm is used to cluster the data. Afterward, K-Means and C4.5 are used to compare classification accuracy and time performance. KDD99 is a well known data set and is used to test if this approach could work on other network data sets.

3.1.1 Data Set.

The data set used to evaluate the approach introduced in this article is the KDD99 data set. This data set was introduced for the 1999 KDD Cup challenge to accurately classify network data to a given class of attack or normal traffic (KDD Cup 1999). The data set consists of both normal and attack traffic classes, with the attack classes making up the majority of the cases. In total, there are 23 classes and 41 features in the original KDD99 data set [52]. Later, this data set was transformed into to NSL-KDD99 which solved some issues regarding redundant samples and the over training of classifiers. Dimensions of this data set include measures such as the protocol type, service, server error rates, and byte counts. The values types in the data are nominal, discrete, and real. Attack types include back dos, buffer_overflow u2r, ftp_writer2l, guess_passwd r2l, imap r2l, ipsweep probe, land dos, loadmodule u2r, multihop r2l, neptune dos, nmap probe, perl u2r, phf r2l, pod dos, portsweep probe, rootkit u2r, satan probe, smurf dos, spy r2l, teardrop dos, warezclient r2l, and warezmaster r2l.

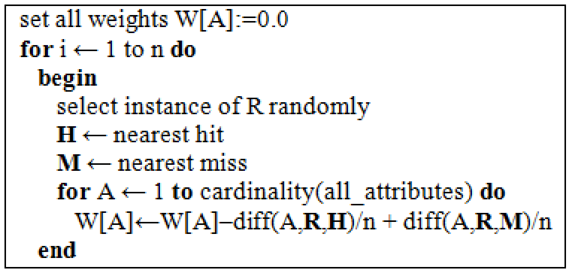
Table 3.1: Number of Samples, NSL-KDD99 Classes

Item	Type	Count
1	normal	67343
2	neptune	41214
3	werezclient	890
4	ipsweep	3599
5	portsweep	2931
6	teardrop	892
7	nmap	1493
8	satan	3633
9	smurf	2646
10	pod	201
11	back	956
12	guess_passwd	53
13	ftp_write	8
14	multihop	7
15	rootkit	10
16	buffer_overflow	30
17	imap	11
18	warezmaster	20
19	phf	4
20	land	18
21	loadmodule	9
22	spy	2
23	perl	3

Table 3.1 shows the breakdown of the sample sizes per class in the NSL-KDD99 data set. In this article, only classes with 20 or more samples will be considered due to the difficulty that classifiers have in distinguishing between small sample sizes. After this modification, there are 14 classes to process and 125,901 samples in all. This is a relatively small decrease as the original sample size of the KDD99 data set is 125,973.

3.1.2 Feature Selection.

Before a classifier is created, only the NSL-KDD99 data sets significant features are selected. The goal is to reduce the number of features to be classified so there are fewer features to process without diminishing the accuracy. The ReliefF algorithm [32] was used to select the best data set features for use by the classifier.



```
set all weights W[A]:=0.0
for i ← 1 to n do
  begin
    select instance of R randomly
    H ← nearest hit
    M ← nearest miss
    for A ← 1 to cardinality(all_attributes) do
      W[A]←W[A]–diff(A,R,H)/n + diff(A,R,M)/n
  end
```

Figure 3.1: Relief Algorithm Pseudocode [32]

The Relief algorithm, shown in Figure 3.1, is the original algorithm that ReliefF is based upon. The Relief algorithm requires an input n for the number of instances to randomly select and updates the weight values associated with each feature by choosing the two nearest neighbors [30]. One of these two nearest neighbors is selected inside the class of the randomly selected instance while the other is the nearest outside the class. The ReliefF algorithm slightly modifies the original Relief algorithm by using the nearest k neighbors, where k is specified as an input, in and outside the class. Instead of choosing one nearest neighbor, it chooses k neighbors. This modification reduces the number of random selections of instances needed and makes it more robust [32]. ReliefF also allows for any number of classes [32].

The primary parameters in WEKA for the ReliefF algorithm are the number of neighbors and the number of instances. The number of neighbors parameter is used to select a given number of neighbors to search in the algorithm. The neighbors include those samples in the randomly selected samples class as well as the nearest neighbors in the other classes. As a result, the parameter must be carefully selected so the number does not go above the smallest total amongst the classes. The number of instances parameter defines the number of instances to select. For each random instance, the ReliefF algorithm updates the weights of each feature. The weights are similar to a scoring feature used to rank the best features and updated according to the equation in Equation (3.1).

$$W[A] := W[A] - \frac{diff(A,R,H)}{n} + \sum_{C \neq class(R)} \frac{[P(C) * diff(A,R,M(C))]}{n} \tag{3.1}$$

Equation (3.1) is the ReliefF weight update equation. In this equation, the parameters are: W (weight), A (attribute/feature), R (random instance), H (nearest hit inside the class), M (nearest miss in the class), and n (total number of instances). This equation is for one instance of k and is repeated the number of times k is set with the exception of excluding the previously used nearest hits and misses.

In feature selection, a greater number of instances will produce the most accurate weights due to the fact that they will approach their steady state values after enough random samples are selected. After the features are selected, a classifier will be built using those features.

3.1.3 Classification.

The data is broken up using a 10-fold cross validation to train the classifiers using the proposed classification method. The classification is performed using both the K-Means [39] and C4.5 algorithms. The K-Means algorithm operates by using a given a set of initial clusters and then assigns each point to one of them. Next, each centroid of the cluster is replaced by the mean point on their respective cluster [17]. Figure 3.2 displays the steps of the algorithm.

```
1. Place $k$ points into the space represented by the objects that are being clustered. These points represent initial group centroids.
2. Assign each object to the group that has the closest centroid.
3. When all objects have been assigned, recalculate the positions of the $k$ centroids.
4. Repeat Steps 2 and 3 until the centroids no longer move. This produces a separation of the objects into groups from which the metric to be minimized can be calculated.
```

Figure 3.2: The pseudocode for the K-Means algorithm is shown below [39]

```
1. Check for base cases
2. For each attribute a, find the normalized information gain from splitting on a
3. Let a_best be the attribute with the highest normalized information gain
4. Create a decision node that splits on a_best
5. Recurse on the sub lists obtained by splitting on a_best, and add those nodes as children of node
```

Figure 3.3: Pseudocode for the C4.5 algorithm [47]

The K-Means classifier from WEKA is chosen for this application due to its ease of implementation. The number of clusters to be chosen is based on the number of classes in the data set. The goal is to have each class mapped to a cluster using the best features selected by the ReliefF analysis. The trained classifier will then classify a set of test samples and the accuracy will be measured.

Pseudocode for the C4.5 algorithm [47], also known as J48 in the open source Java implementation in WEKA, is shown in Figure 3.3 [33]. These algorithms are used for feature selection and classification which results show promise in abstracting data.

3.2 Results

The results demonstrate a faster classification can be performed without much loss in accuracy. After using ReliefF for feature selection, the best 10 features are clustered using Expectation Maximization (EM) algorithm. The two classifiers used, K-Means and C4.5, show that both benefit in performance with a minor loss in accuracy. This is a tradeoff to be able to classify network attacks in real-time.

3.2.1 Feature Selection Results.

Using the WEKA ReliefF implementation, several runs/experiments were performed. The neighbor number k, and number of instances, n is adjusted. Table 3.2 and Table 3.3 show the top 10 best attributes (features) across multiple values of k (number of classes). A feature is considered better if it has a higher correlation to the value being classified.

All samples in this data set was used to randomly select as an instance. For the number of neighbors k, values of two and ten were used. Two is the smallest total quantity for a class. This makes it reasonable to make two the most likely accurate weight. Ten was also used as the number of neighbors with the result having the same top 10 features except with a different order among them. These simulations were rerun with a much smaller set instances chosen at random to see if the same 10 features were selected again.

Table 3.2: Top 10 Best Attribute from ReliefF Using WEKA with n = 5000

k = 10

Feature Number	Feature Name
3	service
38	dst_host_serror_rate
4	flag
25	serror_rate
26	srv_serror_rate
12	logged_in
39	dst_host_srv_serror_rate
29	same_srv_rate
33	dst_host_srv_count
34	dst_host_same_srv_rate

k = 5

Feature Number	Feature Name
3	service
38	dst_host_serror_rate
4	flag
25	serror_rate
26	srv_serror_rate
29	same_srv_rate
12	logged_in
39	dst_host_srv_serror_rate
33	dst_host_srv_count
34	dst_host_same_srv_rate

k = 2

Feature Number	Feature Name
3	service
4	flag
38	dst_host_serror_rate
29	same_srv_rate
25	serror_rate
26	srv_serror_rate
33	dst_host_srv_count
12	logged_in
34	dst_host_same_srv_rate
39	dst_host_srv_serror_rate

Table 3.3: Top 10 Best Attribute from ReliefF Using WEKA with n = 1000

k = 10

Feature Number	Feature Name
3	service
38	dst_host_serror_rate
4	flag
25	serror_rate
26	srv_serror_rate
12	logged_in
39	dst_host_srv_serror_rate
29	same_srv_rate
33	dst_host_srv_count
34	dst_host_same_srv_rate

k = 5

Feature Number	Feature Name
3	service
38	dst_host_serror_rate
4	flag
25	serror_rate
26	srv_serror_rate
29	same_srv_rate
12	logged_in
39	dst_host_srv_serror_rate
33	dst_host_srv_count
34	dst_host_same_srv_rate

k = 2

Feature Number	Feature Name
3	service
4	flag
38	dst_host_serror_rate
29	same_srv_rate
25	serror_rate
26	srv_serror_rate
33	dst_host_srv_count
12	logged_in
34	dst_host_same_srv_rate
39	dst_host_srv_serror_rate

Compared to the previous three figures, there are no changes in the features displayed after changing *n* from 5000 to 1000 instances. There are, however, some differences in the ordering of the rankings when 1000 random instances are selected, but the same features are all represented in the top ten.

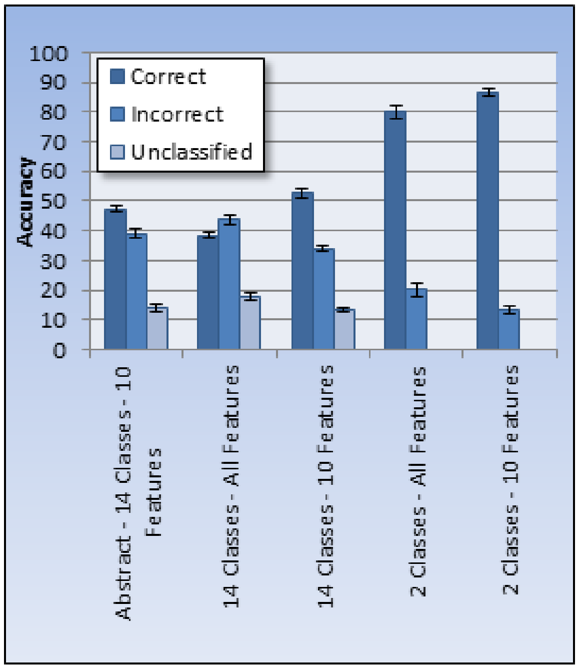


Figure 3.4: Results from K-Means varying number of classes with and without feature selection.

3.2.2 Classification Results.

The same number of entries (125,973) was used for the classification results. The first classifier used for clustering was K-Means. There were five configurations of the data set used to test for classification accuracy. Two configurations were using all features while choosing 2 and 14 clusters. Afterwards, the ten features found by the previous ReliefF calculations were used with 2 and 14 clusters chosen for the next two configurations. The last configuration was the abstraction of the ten features with 14 clusters chosen.

Abstraction of this data set was created by using the EM algorithm. The result of this process created clusters for each feature to classify based on the cluster identifier information. The K-Means algorithm for classification worked best for the 2 classes configurations with the results illustrated in Figure 3.4. There were unclassified samples for the configurations with exception of the 2 classes configurations. The reason for this is due to low confidence of classifying some samples. When reduce to 2 classes, the samples could be classified with greater confidence. However, switching to the C4.5 algorithm the result is dramatically boosted by the use of the feature abstraction as shown in Table 3.4, Table 3.5, and Table 3.6.

Comparing Table 3.4 and Table 3.6 shows that the abstraction system classifies the attack type slightly worse on all 14 classes (95% vs. 97%) in the abstracted data-set with a significantly reduced cluster-based data-set. However, as indicated by Table 3.7 the classifier also acts far quicker (347% faster with K-Means and 48% faster using C4.5) on the simplified cluster-based data set which can be critical in a time-sensitive application. The difference in performance for abstracted versus not abstracting listed in Table 3.7 is found by comparing the total time to classify all samples in the data set. Confusion matrices and summary statistics can be found in Appendix A.

Table 3.4: Results from C4.5 Using 14 Classes

	C4.5 All Features	C4.5 Ten Best Features
Correct	99.7967%	97.4791%
Incorrect	0.2033%	2.5703%
Unclassified	0%	0%

Table 3.5: Results Using C4.5 and 2 Classes

	C4.5 All Features	C4.5 Ten Best Features
Correct	99.7817%	98.487%
Incorrect	0.2183%	1.513%
Unclassified	0%	0%

Table 3.6: Results Using C4.5 and 14 Classes on Abstract Data Transformed from the EM Algorithm C4.5 Ten Best Features

	C4.5 Ten Best Features
Correct	95.0811%
Incorrect	4.9189%
Unclassified	0%

Table 3.7: Time Comparison Using Abstract Features Vs. Normal Feature Set 10 Features, 14 Classes K-Means C4.5

10 Features, 14 Classes	K-Means	C4.5
Abstracted	5.67	3.69
Normal (in seconds)	25.38	5.49
Improved %	447.62%	148.78%

3.3 Summary

This work introduced a new concept of using auto-clustering with the Expectation Maximization algorithm to significantly simplify the feature-set of network traffic along with the use of auto-feature extraction to reduce the number of features. As a result, using the K-Means algorithm the system was able to improve classification speed over 14 classes by over 347%. The C4.5 algorithm has speed increases 48%, which these two increases clearly indicates this method can be effectively used to improve classification accuracy by a significant margin. With the methodologies described in this paper, administrators are able to assess a networks health and status more easily. By utilizing the uniqueness of various features in a network, they can be clustered and evaluated to recognize a cyber-attack. There are multiple avenues for future work using auto clustering and feature selection in networks. A more complete version of the KDD99 data set could be used to improve classification results. Success would depend on available computer resources. Experiments using a k value of 10% resulted in long algorithm run-times. Another interesting area for future research would be implementing a larger variety classification and feature selection algorithms. For example, the K-Means algorithm is relatively simple and easy to implement, but suffers from two drawbacks. First, it is often slow and expensive when used on large data sets and can be sensitive to the initial clusters selections [17] which is a stochastic process. Performing a comparative analysis on various classification and feature selection algorithms would provide an indication of which algorithms are more successful in detecting bad network traffic. Furthermore, there are many parameters associated with the algorithms used in this paper and a study to understand how they compare to the current results when undergoing a wider range of experiments would prove useful. Finally, a test benchmark could be constructed to create more realistic data sets than KDD99. This opens up the ability to perform simulated attacks to test the classifier. More accurate classifiers lead to better and more accurate planning and response systems.

After classifying the attack type, a method for understanding the rewards for a given network is needed as well for implementing an automated planner in the cyber domain.

IV. Learning Rewards using Neural Networks

Identification of the best computer network state is often dependent on the preferences of the network's operator or users. To defend a computer network, knowledge of which network state is better for the operator is key to a successful defense. By implementing a multilayer neural network, this chapter investigates if capturing preferences through samples is viable in estimating reward estimates to use in the UCT planning algorithm. This portion of the research explores the viability of collecting and finding reward values for a general network.

4.1 Methodology

The idea of this chapter is to implement a neural network to learn the network operator's reward preferences. The neural network is trained using a set of samples obtained from observation of the network operator. After training is completed, the neural network is able to output estimates of network preferences based on a model learned from training. If the preferences are changed, a new set of samples is needed to create a new model to capture those changes. Several neural network configurations must be examine to understand what is suitable in order to learn the preferences of network operators. Parameters of the configuration include the number of layers, number of neurons per layer, connectivity of the layers, and the number of output neurons. To demonstrate how a neural network can be used to collect reward preferences, a manageable size model is created to capture data to be used by the neural network.

4.1.1 Network Model.

The network model was created with a size of 8 networked nodes. Each node represents a typical part of a computer network. The types of nodes in this topology are shown in Table 4.1 as well as the topology of the network model shown in Figure 4.1. This

Table 4.1: Nodes of Model Network

Type	Node Count	Initial Weight Value	Changed Weight Value
Router	1	0.6	0.25
Ubuntu	1	0.02	0.02
CentOS	1	0.02	0.04
Windows	2	0.02/0.04	0.04/0.04
Web Server	1	0.075	0.05
File Server	1	0.075	0.06
E-Mail	1	0.15	0.5

is an arbitrary configuration with enough nodes to represent most assets typically found on a computer network. The model can also be easily extended to larger networks or other network configurations.

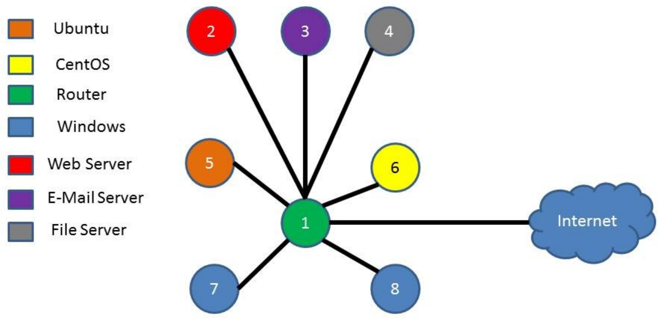


Figure 4.1: Network Topology

Each node is a binary value in which it is either operable or inoperable. Having 8 nodes and binary values for each, 2^8 possibilities exist for inputs. This creates a training and testing set of 256 input values. For each of these 256 samples, the desired output has to be created.

The output is normalized for several reasons. First, it extends easily to asking the operators on a scale from 0-10 how the network state is based on their preferences. This allows a straight forward translation from their data to data that can be used by the neural network. The other reason is that the neural network activation function is bounded by -1 to 1 so translating to these bounds is not difficult.

Calculating the desired output values to match the 0 to 1 scale is done by assigning weights to the network topology that sum up to 1. Network preferences from the operator can be gathered using a similar scale of 0-10 which translates to the normalization without issues. This is done for the model data which is a good initial set of data until the network preferences can be learned from the operator. This work explores updating the initial preferences to the operator's desired output.

4.1.2 Neural Network Configuration.

Approximately 20 neural network configurations are sampled within the parameter range. To limit the number of simulations, the parameters, layers and number of neurons per layer, is kept below 10. Each configuration is tested by examining the Mean Squared Error (MSE) during the 10 folds and observing its performance. Many configurations are able to train to very low MSE. Training a neural network by using back propagation calculations causes tradeoffs on the number of layers versus the time performance to fully train [54] [49]. As a result, the smaller configurations with the least amount of layers are used. Each configuration is a fully connected neural network. The number of hidden layers and the number of neurons that each layer equally contains are found by testing multiple configurations. The output layer is automatically added to each configuration along with the number of neurons for the output layer. For the output layer, the number of neurons is determined by the dimensions of the desired data. For this work, the number of dimensions is one since we use only one number to represent the operator's expected reward.

The Learn Rate (LR) parameter is within the bound [0,1] in order to have a steady learning process. For this neural network, the LR is an exponential decay function expressed as:

$$LR = 0.2 * e^{-n/1000} \tag{4.1}$$

The LR begin at 0.2 and decays over *n*, the epoch iteration, until it reaches 1000 epochs. For each epoch, all the training samples in the training folds are used to update the weights at the learn rate for that epoch iteration.

For the neural network activation function (out), the hyper tangent function is used:

$$out = tanh(\frac{2}{3} * x) \tag{4.2}$$

The output range of these functions is [-1,1]. Another feature of using this function is that it allows for a simple derivation used in the back propagation of the multilayer neural networks [29] [41]. The derivative is expressed by:

$$d(out)/dx = (1 - tanh^2(\frac{2}{3} * x) \tag{4.3}$$

The weights of the neural network are initialized using an uniform random distribution with the range of [-0.4,0.4]. The weights are chosen because of the linear shape of the activation function at the center of x axis.

After testing multiple neural network configurations, a neural network with two hidden layers, each with six neurons per each layer, is chosen. Observations indicated that several configurations had similar performances but trained slower the more layers in the neural network configuration. The more number of layers in the neural network, the longer the training session takes. Empirical evaluation revealed that two layer network with six neurons trained faster with similar performance learning the preference functions.

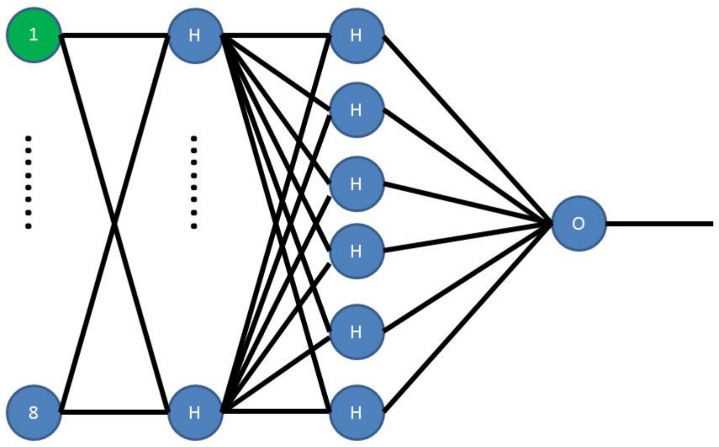


Figure 4.2: Final neural network configuration used.

4.1.3 Error Calculations.

The error metric for determining the neural network performance is Mean Squared Error (MSE). During training, the squared errors of each sample are accumulated. Afterward, the total is divided by the number of training samples. The convergence criteria for learning the neural network uses this error metric. However, the MSE is not used to update the weights of the neural network. The difference of the desired output and the neural network output is the error for the weight update calculations. MSE is calculated to track performance for the training, validation, and testing.

4.1.4 Using Sample Data to Train the Neural Network.

Once the sample data set was created, a method to divide the samples into different training, validation, and testing sets is required. To train the neural network, the diagram in Figure 4.3 shows how the samples are divided into each fold. Three fourths of the sample data is used for training and the remainder is left for testing. These portions are created by random selection. To ensure an even representation of all the output, the portions randomly select among thirds of the ordered 256 samples. This reduces the likelihood of a bad random selection that might not capture the complete desired range. With the $\frac{3}{4}$ portion, a k-folds cross validation is used with k (number of folds) equal to ten. After the 10 folds are created from the $\frac{3}{4}$ portion, one fold is used for validation and the rest are used for training. The convergence criteria for the training is to stop when MSE = 0.01. At this point the training is considered complete and testing separated samples on the neural network begins.

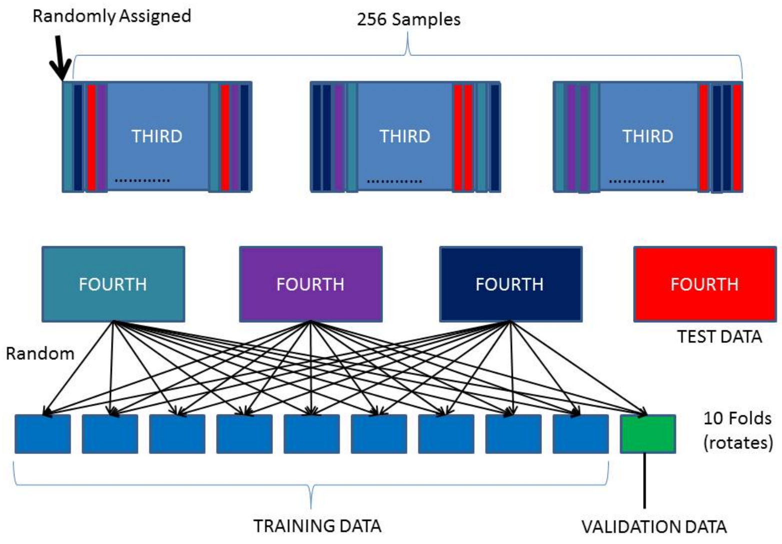


Figure 4.3: Creating Training and Test Data Sets

4.1.5 Testing the Neural Network with Separated Samples.

A $\frac{1}{4}$ portion of the 256 samples is set aside for testing. This reserved data set is used to validate that the neural network output is matching the desired reward preferences. The testing sample portion is passed through the neural network each epoch to see how the performance of the neural network is with respect to the testing samples. One aspect that has to be considered is noise in the collection of the samples.

4.1.6 Noise in Sample Data.

One area of concern when gathering preferences is operators inconsistency. If the operator gives preferences not corresponding to their overall expected preference model, then the neural network could be negatively impacted. Because of the weak operator consistency, zero mean Gaussian noise is added to the training desired output in order to research possibility of this occurring. If the model changes, the system will have to update the model.

4.1.7 Updating the Model.

The preceding sections introduce methods detailing how an initial data set of preferences can be constructed. However, the neural network should be able to learn new preferences as they are observed. This work shows what happens when the data set reaches the halfway point of old and new preferences and alternatives to enable additional preference functions. The results demonstrate how it impacts the neural network and changes that have to be made.

4.2 Results

The chosen neural network configuration performed well with a MSE close to zero. Figure 4.4 shows how much squared error occurs during the training phase. It appears to be noisy but this is to be expected due to randomization of the order of the training samples for each epoch. Each epoch is 172 training iterations. By looking at Figure 4.3, we see that every 172 iterations the overall squared error is decreasing. By the fifth epoch, the squared error is small.

While Figure 4.4 shows what the squared error looks like during training, Figure 4.5 shows the overall training process with all 10 Folds. This figure highlights how each fold is shuffled and the variation between training sessions on the different 9 folds each time. Interestingly, most of the training sessions went to a low MSE by the fifth epoch.

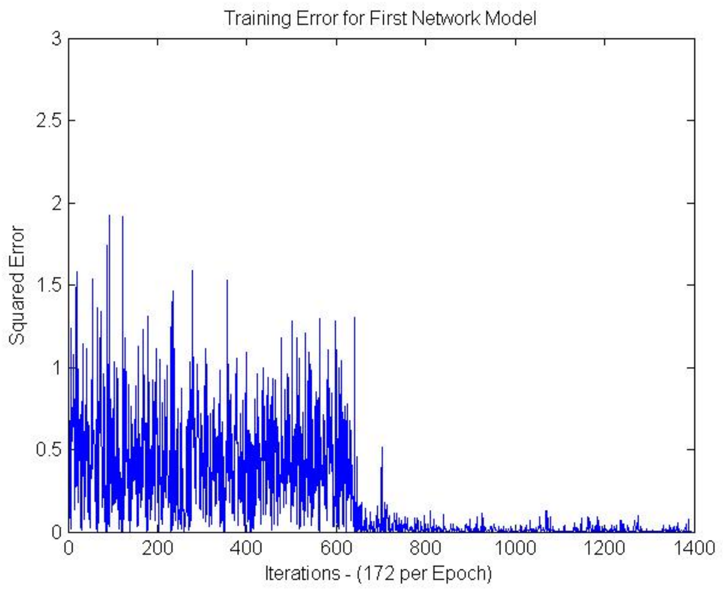


Figure 4.4: Squared Error over Training Iterations

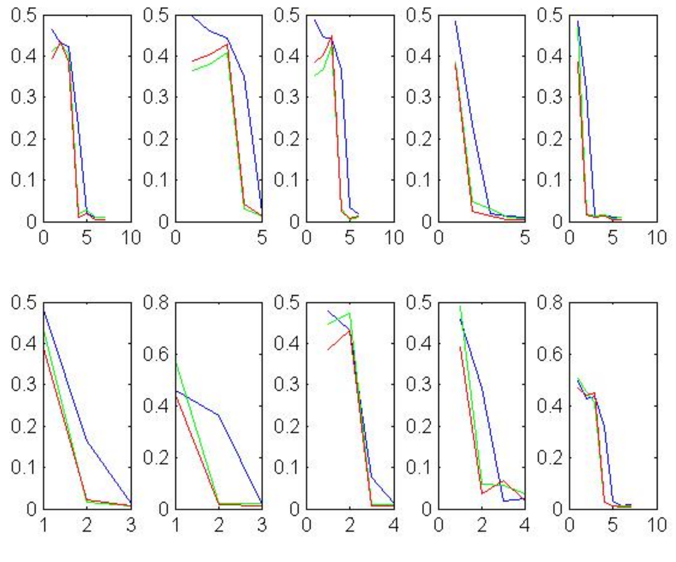


Figure 4.5: MSE vs Epochs For 10 Folds (axis labels hidden to save space; same axis labels as Figure 4.6)

Figure 4.6 shows the average of the plots in Figure 4.5 for a clearer overall picture of how the neural network performs in general for the initial preference data set.

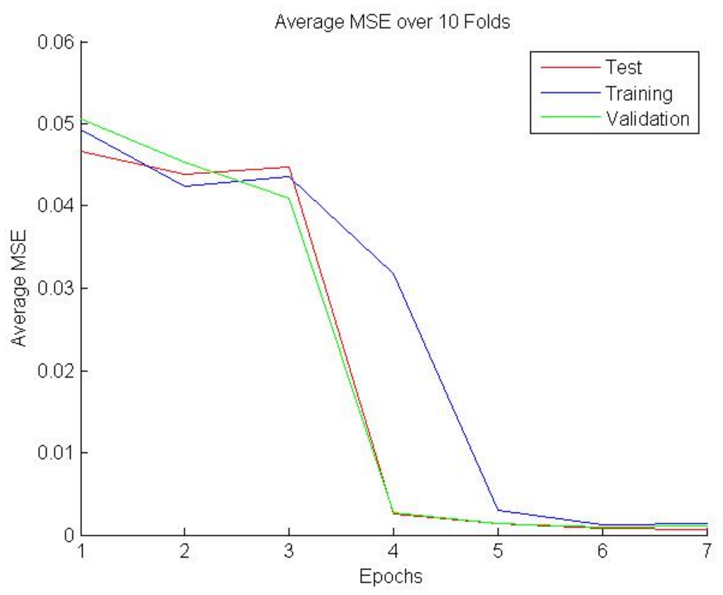


Figure 4.6: Average MSE Across 10 Folds

Results from adding the zero mean Gaussian noise, with a variance of 0.05, to the desired output of the samples is shown in Figure 4.7, and changing the variance to 0.1 is shown in Figure 4.8. These two figures demonstrate the effect the variance has on the neural network. Initially, with a 0.05 variance the neural network does not appear to be effected greatly, however, after the variance exceeds 0.1, the testing and validation MSE starts to increase.

Another aspect to learning user preferences is the ability to learn new preference models. Mixing the data set with half old and half new preferences is done to validate the performance of the neural network when it reaches the halfway point of learning the new preference function. Figure 4.9 illustrates what happens at this point in learning the

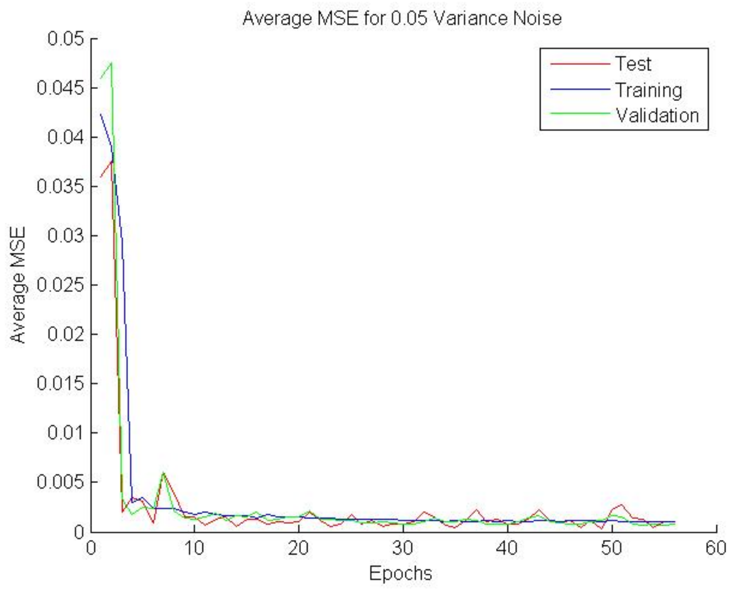


Figure 4.7: Noise - 0.05 Variance Zero Mean Gaussian

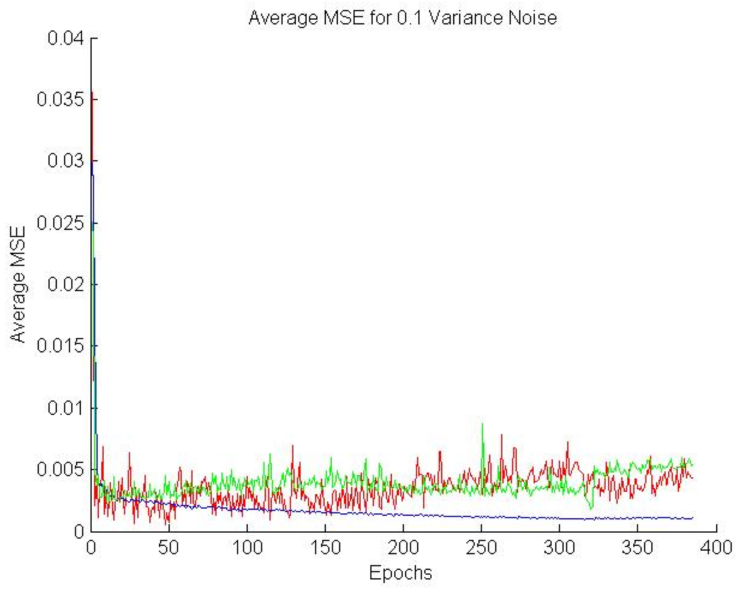


Figure 4.8: Noise - 0.1 Variance Zero Mean Gaussian

new preference. With only one output neuron, the neural network cannot handle outputting the two preference functions. As the learning takes place progress eventually plateaus with regard to MSE because it is already doing the best it can for both preference functions. The findings of this result encouraged the addition of another output neuron added for the second preference function to be learned.

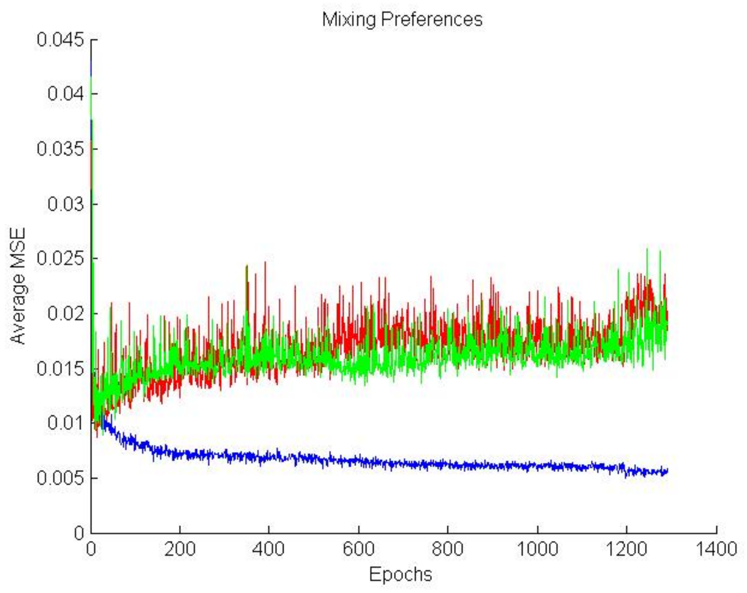


Figure 4.9: Half Old and Half New Preferences Training

The outcome of adding a second output neurons is shown in Figure 4.10. The results are similar to how the neural network performed with the initial preference function. It approaches close to zero in about five epochs.

4.3 Summary

Results demonstrate that learning of computer network preferences is feasible with multilayer neural networks. Within certain noise tolerances, the neural network is not effected greatly and some inconsistency when collecting preference data is tolerable. The

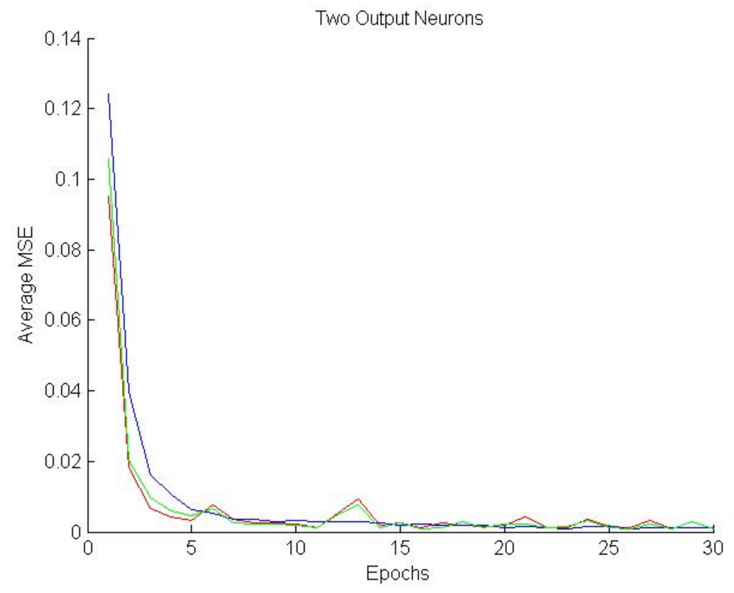


Figure 4.10: Using Two Output Neurons for New and Old Preferences

model can be extended to any network size. This work is a basis to explore how reward preferences can be learned for larger networks. Future research efforts include allowing more than a binary state for each node. An additional state that has an intermediate value of 0.5 can be added to show a degraded state for its node. This adds more complexity to the model but could allow more representative network states greater variation of preferences. The results of using neural networks to learn computer network reward preferences effectively demonstrate that it is a feasible method.

V. Methodology

5.1 Background

Security of computer networks is a major focus for all cyber operators. Cyber operators are often overwhelmed with information during a cyber attack. Because of the speed of cyber attacks, it is hard for operators to defend the network in real-time and sometimes operators are not even aware they are happening until after significant damage has occurred. Thus, the need to create an intelligent system to assist operators in network defense is becoming more pressing as cyber attacks continue to grow and evolve. There are several aspects to this problem that must be addressed by multiple disciplines, including classification, reward learning, and planning.

Intrusion Detection Systems (IDS) are a well established research area in the network defense field. Generally, an IDS analyzes network traffic to determine if an attack is occurring or not. A classifier can be used as a signature based IDS. This research models the planning component as if the system was provided the attack type by the classifier.

Generally speaking, the classifier detects the type of attack by analyzing a sample of the network's traffic. There are several ways to classify the network data [13] [56] including clustering by K-Means [39] and Decision Trees, like C4.5 algorithm [47]. K-Means classifies by converging on the statistical means of *k* number of clusters. Decision Trees classify by constructing several paths which are routed by learned boundaries for given features that eventually lead to a classification based on the values of the sample for each feature. A signature based IDS detects intrusions in order to alert the operator by classifying the type of attack. Formulating set of actions to respond to the attack is still left up to operator to do in real-time. To assist with this issue, an automated planner to formulate a plan given an allotted time is investigated in this thesis.

There are several kinds of automated planners such as state, plan, plan-state hybrid, and MDP based [22]. The focus for this portion of the problem is to use the UCT algorithm [31], a MDP based planner, for the automated planning. UCT is similar to a Monte-Carlo simulation but has a more targeted search which is key for a fast planning algorithm. Formulating a plan to find the best network state is accomplished by performing a sequence of actions and is the role the UCT automated planner performs in the system.

5.2 Problem Definition

5.2.1 Goals and Hypothesis.

The primary goal of this research is to evaluate a system designed to assist cyber operators in responding to a cyber attack faster by automating processes that the operator would not be able to do in real-time. This system is not likely to defend the network against unknown types of attacks since the classifier depends on trained data in order to detect attacks. The system is created to determine its effectiveness against known attacks in real-time by providing a plan. Using data from known attacks guides what actions to take and determines whether the plan formulated is effective. To reach this goal, sub goals need to be achieved because the system has many components.

The IDS or classifier component of the system must detect if an intrusion is occurring. It has to know whether traffic is normal or hostile. The system is dependent on the effectiveness of the IDS but for the purposes of the goal classifier integration is excluded to focus on a real-time planning system.

Whenever the IDS classifies the type of attack, the first sub goal of the effort is to determine whether it can do so accurately. The classifier is given a network data sample to classify to determine what type of response to plan. The classifier must classify the type of attack traffic accurately for the plan to formulate accordingly. The type of attack determines what actions are available. In this thesis, the real-time classifier portion is not combined with the planning portion. However, both are researched to examine the feasibility of developing a classifier and planning combination to create a real-time cyber defense system. The final goal is to determine the ability of the planner component to formulate effective plans.

The sub goal of developing a cyber attack response plan is to test whether the best network state results from the plan's suggested response actions. To reach this sub goal, the best network state is represented in a manner that captures the most desirable attributes of a network. These sub goals are met and performed in real-time to achieve the primary goal. With better automation provided by planning and classifying, the operator can successfully defend against attacks faster, and in real-time.

5.2.2 Approach.

To achieve the research goals for this problem, components are built for each of the sub goals. To detect the type of network attack, a component is developed to read network data samples and classify them accordingly. The classifier must be accurate to minimize errors in creating a plan. It also must perform detection quickly for the planner to have time to run as well. The approach used is to test a variety of classifiers on the NSL-KDD99 data set [52] which is the most recent published network data samples.

Although this data may not be as relevant in today's network, the primary purpose of this component is to classify an attack so the planner can formulate a plan for that specific attack. This is more realistic because these attacks are better understood than more recent attacks and having several actions to choose from to create a plan is more robust in demonstrating planning. As newer data is available, the system can be trained to accommodate the newer style attacks.

Previous chapters have shown, pre-clustering data for classification has uses in detecting anomalies for unknown attack types. In addition to using the NSL-KDD99 data set to train classifiers, the data set is used to determine if the number of features used in the classifier can be reduced. This feature selection method uses the ReliefF algorithm to find the best features [32, 33]. By reducing the number of features to examine, the classifier can be faster. The downside is that accuracy is likely to decrease. However, this decrease is not likely to be significant enough to affect results.

Widely-used classifiers are used: the clustering by K-Means, C4.5, and Neural Networks [24]. These are the most common classifiers and yet they are very different from each other. Countless variations and other methods can be used as classifiers. To limit complexity, these three which are well defined and understood are used. These classifiers are compared for accuracy and performance and coupled with the planning system.

The planning system is created using UCT as the search method. UCT was chosen as a result of researching different automated planners and fast methods. Another key part of the planner is representing the network state. Having a suitable network state gives an indication of which is the best plan. The reward for performing a plan is based on the network state. The first approach to representing the network state is to have priorities associated with parts of the network and identifying if those parts are operable.

5.3 System boundaries

The Network Defense Planning System (NDPS) is shown in Figure 5.1. The component under test is the planning component. This component runs on a desktop computer and models intrusion information an IDS/classifier would send. The real-time operation of the NDPS is the focus of this effort. The IDS signals whether an intrusion is occurring and if so initializes the system, then sends it the event information. As a form of scope reduction, this part is simulated. In addition to simulating the classifier information, information from the network is needed to represent the state of the network.

To identify which actions to take, the planner must have a way to represent the reward in order for it to choose a good sequence of actions. Part of this research identifies the most effective method of representing the reward function for determining the best network

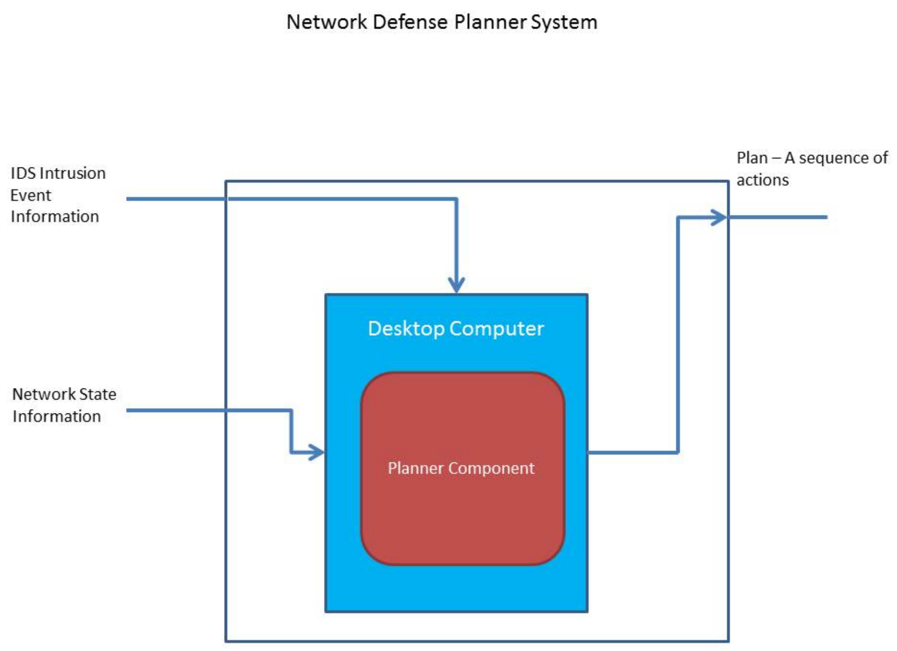

Network Defense Planner System
IDS Intrusion Event Information
Network State Information
Desktop Computer
Planner Component
Plan – A sequence of actions

Figure 5.1: Network Defense Planner System Diagram.

state. The type of information used is the percentage of the network that is still operating. This can be broken down even further to give weights to portions of the network that are more important than others. After the best reward function is determined, the planner can formulate the best plan to handle the current attack.

The scope of this system ends after the plan is formulated. The plan is the output of the system, and the system does not execute the plan. This information may be used by another system or person but that is beyond the scope of this effort.

5.4 System Services

The NDPS provides only one service. That service is to output the best plan which is dependent on the time allowed to formulate the plan. Success or Failure are the two outcomes of this service. A success is defined as whether or not the plan increases the reward based on the reward function defined in the metric section. The failure outcome is the plan decreasing reward or keeping it the same as the initial reward calculation.

5.5 Workload

Information from the simulated IDS classification system is the first workload parameter. NDPS assumes that the IDS always correctly detects attacks and signals operators as needed. The other workload on the system is the time allotted for it to run. Since different attacks take various lengths of time to execute, there may be some extra time to formulate even better responses. This means that the system needs to output a plan within the specified time constraints. A specified interval to output plans shows the progress it makes as it runs. These inputs constitute the workload of the NDPS.

5.6 Performance Metrics

Since the goal of this research is to assess real-time performance, the time to create a stable plan from IDS classification to the formulation of the plan is the primary metric and is shown in Table 5.1.

Table 5.1: Metrics of the NDPS.

Time to Output Optimal Plan
Count Out of Total Runs Optimal Plan is Found
Reward Function

Since all sampling rates are to be performed for each configuration, the timing excludes the time to gather the samples with the exception of one sampling time. This is because NDPS outputs at least one plan whenever requested and that time needs to be considered. The time to sample the plan and output information that is sampled to a file is omitted. The time starts once NDPS gets the generated model file that the IDS would generate and stops once the plan is considered stable. For this timing metric, the plan is considered optimal once it reaches the optimal reward value. The optimal reward value is known by fully exploring the possible search paths of the state-action tree. To determine if a plan is optimal, another metric is needed.

5.6.1 Reward Implementation.

The other metric needed determines the best reward state of the network derived from implementing the plan. This state representation of the network identifies which plan is expected to be best. In this case, the best plan maximizes the reward function

$$R = \frac{\sum_{i=each\ node}^{effected\ nodes} w_i * H_i}{w_{all\ effected}} \tag{5.1}$$

where w_i is the weight given to the ith node, and H_i is an indicator variable that specifies the reward value of the ith node. The variable w_i is assigned so that all the weights sum to one.

In previous chapters, the neural network research demonstrated how the reward values can be obtained by learning from the operators through samples provided. However, since this data would be arbitrarily generated anyway, we employ another method making the research effort more straightforward. The rewards are normalized so that 0 is the starting point where no events are resolved, and 1 indicates that the events are fully resolved. After the events are processed, the total reward for a given state is calculated by parameters of the nodes effected by events. The weights of these nodes are summed and divided based on individual host reward values. Host reward values are also normalized based on the events experienced on that host. The formal equation for this is as follows for hosts:

$$H = \sum_{each\ event}^{all\ events} \frac{(Completeness * ImpactLevel)}{(Sum\ of\ Impact\ Levels * 100)} \tag{5.2}$$

H is the same as mentioned in the methodology section of this paper. Completeness is a normalized value where 0 is nothing fixed and 1.0 is fully resolved (stored as 0-100). The changes in completeness depend on the action and attack type, Table 5.2 lists the changes. There is some ordering but only in the case where the action "Shutdown Host" is selected before an effective action such as "Block IP". In the cases where "Shutdown" is selected before, the completeness value decreases to zero. Impact level allows priority to be assigned to attack types that have a greater impact on network damage. Table 5.3 shows the impact values given the attack type.

Table 5.2: Change to completeness per action and attack type.

	Block IP	Shutdown Host	Disable Service	Reset Password	Patch SQL
Denial of Service	100	+40	+25	No Change	No Change
FTP Brute Force	+90	+40	No Change	+10	No Change
Metasploit	100	+40	No Change	No Change	No Change
POP3 Mailbox Overflow	100	+40	+25	No Change	No Change
SQL Injection	+90	+40	+25	No Change	+10
Port Scan	100	+40	+25	No Change	No Change

These are the arbitrary values used in the configurations to give functionality to the impact level parameter and so all attack types are not treated as equal. The impact and weight of host parameters are more of a placeholder for future research and are fixed for these experiments. Weights for the host are fixed to be equally distributed based on the number of host in order to make all the weights sum to one.

5.6.2 State Abstraction.

The information pulled from the network into the NDPS is important to define. A state in the network is represented by several objects in the NDPS software. The primary

Table 5.3: Impact values of each attack type.

Denial of Service	3
FTP Brute Force	5
Metasploit	3
POP3 Mailbox Overflow	3
SQL Injection	1
Port Scan	5

attributes pulled from the network model file are shown in Table 5.4. This information is used to create a state in the NDPS. Based on what is extracted from the network, parameters such as the rewards and weights are initialized. Event information can be pulled from the network model but are randomly generated scenarios which consist of a mixture of events.

Table 5.4: State abstracted attributes.

Host	IP address
Event	Name / Destination IP address / Port / Source IP address / Impact Level
Action	Name / Options / Host IP address
Network State	List of Hosts / List of all Network Actions

5.7 System Parameters

This system runs on a specified computer in the targeted network. The configuration of this computer does not change to get all the run times on the same baseline. The clock speed of the processor and main memory size is fixed. In addition, the background workload of the computer is minimized by eliminating any non-essential programs while running the experiments. The next parameter is the network structure modeled.

The network structure impacts the planning component because the planner takes actions on a network. For a large network, the planner may need to take more actions to stop an attack. While the other effect the structure has is in the determination of the best network state. One of the most significant attributes for network state is how much of it is operable. The structure of the network also matters due to attacks that focus on different parts of a network. The last parameter is the number of actions available for a given network attack.

The number of actions based on network size is fixed for several reasons. The first reason is to reduce the complexity of the planning so searching through actions that are not applicable to an attack does not occur. The next reason is to get better performance as a byproduct of the reduced branching while searching through only applicable actions. These three parameters are fixed for the NDPS.

5.8 Factors

Factors in the experiments performed come from the workload and system parameters. Table 5.5 shows the factors of the experiments.

Table 5.5: Factors of the NDPS.

Network Randomized Attacks	4 trials
Depth of Plan Formulation	3
c value of UCT Algorithm	$\sqrt{0.1}$, $\sqrt{1}$, $\sqrt{2}$, $\sqrt{3}$, $\sqrt{4}$, $\sqrt{5}$
Sampling Rate of the Plan	Every 10 iterations for 250000 iterations
Network Structure	Small (7 nodes) and Large (12 nodes)

These factors are chosen as a result of their large influence on the primary goal of this research. The Network Randomized Attacks is significant because planner component results vary significantly based on the attack type. Randomized attacks consists of six different attack types that randomly choose applicable targets given the attack type. The randomized attack is generated to have a plan length of three. A depth of three is used in the UCT algorithm because of physical memory constraints. The Sampling Rate of the Plan factor determines when the plan achieves optimality based on the reward function. If the sampling rate is fixed, then the point in which the plan becomes stable may be missed until the next sampling interval. Finally, the next factor is the Network Structure which plays a major role in run time of the system since a planner has to search through more network states as more actions are likely to be needed for a larger network size. Having these factors change illustrates how the results vary best through the experiments.

5.9 Evaluation Technique

The evaluation technique used in this thesis is simulation. The system is able to parse a randomly generated XML file from generated network configurations. However, for this research the XML file is simulated and not from actual network configurations. The plan formulation models actions for an actual network structure as well. This network structure is created using virtual machines networked together on a 100Mbps Ethernet connection via VMware. These nodes have different operating systems that include Windows 7 and Ubuntu. Other node types are Outlook Exchange E-mail and Apache Web servers. The attack types are Denial-of-Service, Port Scans, Metasploit, Brute Force FTP Password, SQL Injection, and POP3 Mailbox Overflow as shown in Table 5.3. The desktop computer specifications performing the NDPS's operations are: Intel Core i7 2.3GHz and 8 GB RAM.

After the randomly generated attack event information is received, the planner component performs simulations on the outcomes of choosing a response sequence of actions. The planner simulation does not actually perform any actions to change the network environment but uses the classification of the network data samples. Executing the plan is out of the scope of the system for this effort.

To verify this system, NDPS is used to fully explore the tree and record the optimal plan for verification. A very large *c* value (approximately $2 * 10^{31}$) is used to always search new paths in the search tree. After the optimal plan is verified, the rewards increases at each plan level are used to collect what iteration and time the experiments found the optimal plan. After the initial full exploration run, 1,000 runs are performed for that randomized attack along with six *c* values and two network sizes.

5.10 Experimental Design

A full factorial design is used. The total number of possible experiments using the factors described is 48000. The 10 iteration sampling rate is used for each configuration. Therefore, no additional experiments are needed for the sampling rate factor. In addition, each configuration is replicated 1000 times to allow better understanding of what variations exist in the system. The reason for 1000 replications is that there is not much variation expected from this algorithm.

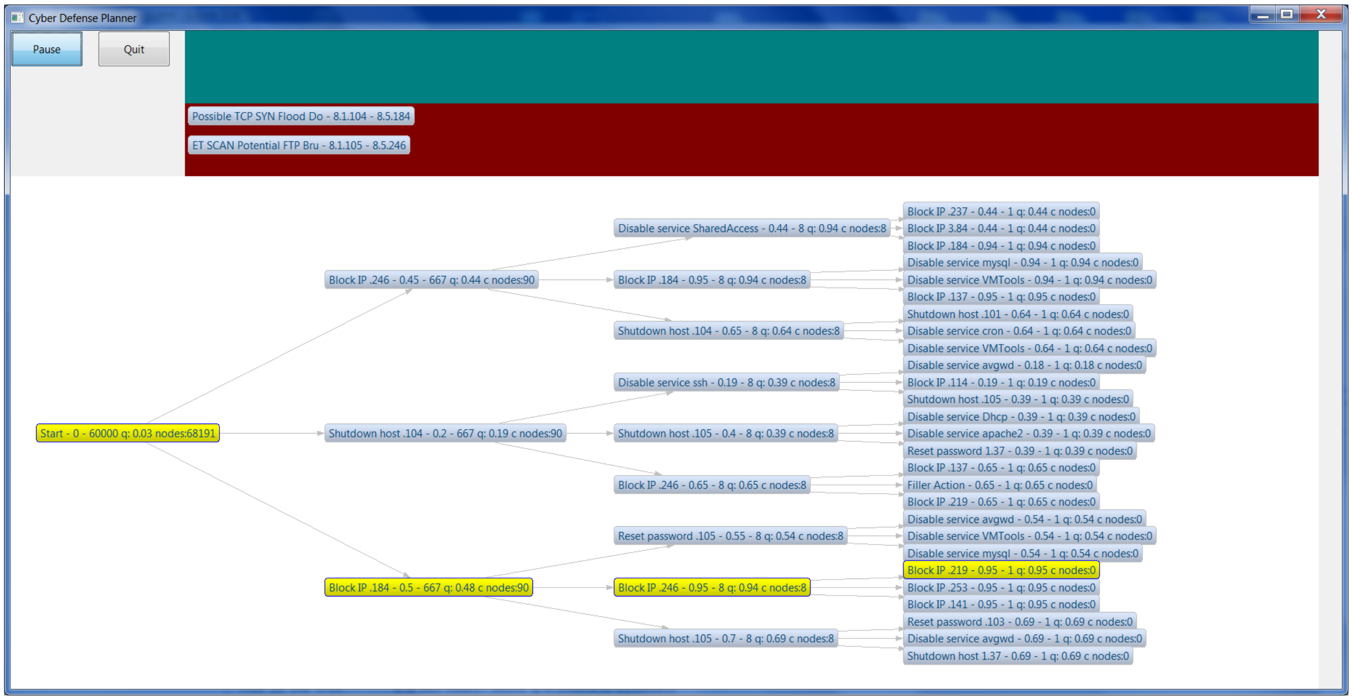
A 95% confidence interval of sufficiently narrow width is expected for the accuracy of classification. The reward state from planning is also a 95% confidence interval with the number of replications performed. These confidence intervals are 95% due to the consequences of being incorrect. The plan could be wrong and the likely worse case scenario is decrease productivity from parts of the network being down.

5.11 Methodology Summary

This research implements a real-time system to assist cyber operators in responding to cyber attacks. The approach is to develop a planner component, so NDPS can provide the needed information to thwart cyber attacks. NPDS scope is based around the IDS/classifier information which is assumed to always be correct. Creating a plan is the only service NPDS provides and the resulting plan is considered either a success or failure depending on the reward function. To evaluate this research, simulations are used to perform the experiments. The testing environment has been described to allow verification. The experimental design is full factorial because the experiments are performed quickly in accordance with the goal requirements. Replications are also included to measure variability with NDPS. To verify the results, extensive searches are used to ensure plan results are correct. This describes the methodology used to perform this research to create a real-time, automated system for cyber operators.

VI. Results

6.1 Interface



Cyber Defense Planner
Pause
Quit
Possible TCP SYN Flood Do - 8.1.104 - 8.5.184
ET SCAN Potential FTP Bru - 8.1.105 - 8.5.246
Start - 0 - 60000 q: 0.03 nodes:68191
Block IP .246 - 0.45 - 667 q: 0.44 c nodes:90
Shutdown host .104 - 0.2 - 667 q: 0.19 c nodes:90
Block IP .184 - 0.5 - 667 q: 0.48 c nodes:90
Disable service SharedAccess - 0.44 - 8 q: 0.94 c nodes:8
Block IP .184 - 0.95 - 8 q: 0.94 c nodes:8
Shutdown host .104 - 0.65 - 8 q: 0.64 c nodes:8
Disable service ssh - 0.19 - 8 q: 0.39 c nodes:8
Shutdown host .105 - 0.4 - 8 q: 0.39 c nodes:8
Block IP .246 - 0.65 - 8 q: 0.65 c nodes:8
Reset password .105 - 0.55 - 8 q: 0.54 c nodes:8
Block IP .246 - 0.95 - 8 q: 0.94 c nodes:8
Shutdown host .105 - 0.7 - 8 q: 0.69 c nodes:8
Block IP .237 - 0.44 - 1 q: 0.44 c nodes:0
Block IP 3.84 - 0.44 - 1 q: 0.44 c nodes:0
Block IP .184 - 0.94 - 1 q: 0.94 c nodes:0
Disable service mysql - 0.94 - 1 q: 0.94 c nodes:0
Disable service VMTools - 0.94 - 1 q: 0.94 c nodes:0
Block IP .137 - 0.95 - 1 q: 0.95 c nodes:0
Shutdown host .101 - 0.64 - 1 q: 0.64 c nodes:0
Disable service cron - 0.64 - 1 q: 0.64 c nodes:0
Disable service VMTools - 0.64 - 1 q: 0.64 c nodes:0
Disable service avgwd - 0.18 - 1 q: 0.18 c nodes:0
Block IP .114 - 0.19 - 1 q: 0.19 c nodes:0
Shutdown host .105 - 0.39 - 1 q: 0.39 c nodes:0
Disable service Dhcp - 0.39 - 1 q: 0.39 c nodes:0
Disable service apache2 - 0.39 - 1 q: 0.39 c nodes:0
Reset password 1.37 - 0.39 - 1 q: 0.39 c nodes:0
Block IP .137 - 0.65 - 1 q: 0.65 c nodes:0
Filler Action - 0.65 - 1 q: 0.65 c nodes:0
Block IP .219 - 0.65 - 1 q: 0.65 c nodes:0
Disable service avgwd - 0.54 - 1 q: 0.54 c nodes:0
Disable service VMTools - 0.54 - 1 q: 0.54 c nodes:0
Disable service mysql - 0.54 - 1 q: 0.54 c nodes:0
Block IP .219 - 0.95 - 1 q: 0.95 c nodes:0
Block IP .253 - 0.95 - 1 q: 0.95 c nodes:0
Block IP .141 - 0.95 - 1 q: 0.95 c nodes:0
Reset password .103 - 0.69 - 1 q: 0.69 c nodes:0
Disable service avgwd - 0.69 - 1 q: 0.69 c nodes:0
Shutdown host 1.37 - 0.69 - 1 q: 0.69 c nodes:0

Figure 6.1: Sample image of GUI for NDPS.

To conduct these simulations using NDPS, a UCT program is implemented using Java. The Graphical User Interface (GUI) of this program is displayed in Figure 6.1 and is used for visualization purposes. The simulations are performed without an interface due to the large number of runs and additional time needed for the interface to process. This sample image shows three important areas of interest. The green area is used to append plans by selecting among the first level of the tree at certain intervals, a feature not used in this research. The red area displays nodes that represent each attack event. Each attack event node has an attack type and displays part of the effected host identifier. The white area displays the UCT search tree over the iterations while it updates. Inside this area, the starting state node is to the left and expands to the right by showing the three best children in terms of number of visits for each node. Each node has a description of parameters for that node. It shows the action leading to that state, the reward value, the number of visits, the *q* value, and the number of unexplored children nodes. The GUI also highlights the best course of action at that iteration which is identified by the yellow coloring.

The string in the node can be summarized using this format:
"(Action Name) (Action Option (Last 4 Characters)) - (Reward Value) - (Number of Visits) q: (*q* value) c nodes: (Number of Unexplored Children Nodes)"

The start node has the format:
"Start - (Reward Value) - (Iterations) q: (*q* value) nodes: (Number of Nodes in Tree)"

Having this interface gives additional understanding of how NDPS works as it performs.

6.2 Constraints

A depth of three is chosen in order for the tree to be fully explored. Increasing the depth causes an exponential increase in the size of the search tree and quickly encounters the memory limits of 8GB. Sizes larger than 8GB of memory would only be able to handle fully exploring a few more levels in depth and branching action size. The limits of memory space is also reached by increasing the cardinality of the action set. The size of the tree is calculated by a^d where the number of actions is a and depth of searching is d. As a result of these constraints, the depth is set to three since full exploration is needed for verification. However, full exploration is not required by UCT so larger depths and number of actions is suitable in real world use. The network model configurations are created so that the number of actions does not increase the tree space past memory limitations.

6.3 Randomization

There are two random objects used in NDPS. The first object randomly generates the attack scenario based on six attack types and random targets. The random seed is generated by the system's millisecond clock time. This seed generation is repeated until the attack scenario has a best plan length of three to resolve. Once this seed is found, it is used to perform the 1000 runs to observe variability in the UCT search. The attack seeds recorded in Table 6.1 and can be replicated in this program.

Table 6.1: Random Seeds for Attack Scenarios.

Small Network	168724314686619, 250415302950019, 305745229196489, 205437984853276
Large Network	458188150037820, 208107068068252, 466685700278999, 29905165582332

The other random object is used by the Monte-Carlo portion of the UCT search. This random object generates the default seed using system millisecond clock time. These two objects ensure sufficient randomization by using the sequence of two objects as opposed to using the first few numbers of sequences from creating multiple objects each iteration.

6.4 Attack Scenarios

Four attack scenarios are randomly generated for each of the two network sizes. Table 6.2 shows the attack information for all the scenarios. Even though randomized, the first two large scenarios generated the same attack except each one was from a different source. This means the results of the first two large scenarios should be comparable to each other, if not the same. Table 6.3 demonstrates the optimal plan needed to resolve each scenario as well as the respective reward value increments. These optimal actions are needed to calculate the plan length when randomly generating an attack scenario. All attack types except Brute Force FTP Password and SQL Injection have one optimal action. Brute Force FTP Password and SQL Injection have two optimal actions needed to resolve the attack event. The optimal action count for a given attack scenario is used to calculate if the optimal plan has a length of three. If the scenario generated does not have a optimal plan length of three to match the depth, then the scenario is randomly generated repeatedly until the criteria is met. Optimal actions are considered optimal according the the reward calculations. They are not considered optimal per attack type in a realistic situation. Although, the actions are reasonable to take against such attack types. Proving that the actions are optimal per attack type is a considerable task, and the research community continuously updates optimal responses as the attack types evolve. The most current optimal actions that resolves a certain attack could be placed in this program along with proper reward considerations.

Table 6.2: Attack Scenarios Randomly Generated (slashes represent separation of attacks on a host).

Scenario	Events
Small Scenario 1	POP3 Mailbox Overflow / Brute Force FTP Password
Small Scenario 2	Two Different Denial-of-Service / Denial-of-Service
Small Scenario 3	Port Scan / Metasploit & Denial-of-Service
Small Scenario 4	Denial-of-Service / POP3 Mailbox Overflow & POP3 Mailbox Overflow
Large Scenario 1	Denial-of-Service / Brute Force FTP Password
Large Scenario 2	Denial-of-Service / Brute Force FTP Password
Large Scenario 3	Metasploit / Brute Force FTP Password
Large Scenario 4	Port Scan / SQL Injection

6.5 Network Configurations

The small network configuration is made up of six hosts and one firewall host. These seven machines create the action set that the algorithm evaluates and searches through. In this set of actions, many actions have no impact on certain attack types. This offers a

Table 6.3: Optimal Plan for Attack Scenarios Randomly Generated.

Scenario	Best Plan Found	Reward Increments
Small Scenario 1	Block IP → Block IP → Reset Password	0.5 → 0.95 → 1
Small Scenario 2	Block IP → Block IP → Block IP	0.5 → 0.75 → 1
Small Scenario 3	Block IP → Block IP → Block IP	0.5 → 0.75 → 1
Small Scenario 4	Block IP → Block IP → Block IP	0.5 → 0.75 → 1
Large Scenario 1	Block IP → Block IP → Reset Password	0.5 → 0.95 → 1
Large Scenario 2	Block IP → Block IP → Reset Password	0.5 → 0.95 → 1
Large Scenario 3	Block IP → Block IP → Reset Password	0.5 → 0.95 → 1
Large Scenario 4	Block IP → Block IP → Patch SQL	0.5 → 0.95 → 1

balanced approach; while some actions are not relevant to one attack, they may be relevant to others. Some actions effect an attack but may not be optimal. One example is shutting down a machine versus blocking the malicious network traffic. This mix of actions offers a diverse action set to explore.

Table 6.4: Attack Scenarios Randomly Generated.

Scenario	Action Count	Host Count	Event Count	Total Node Count
Small Scenario 1	54	7	2	160435
Small Scenario 2	55	7	3	169456
Small Scenario 3	55	7	3	169456
Small Scenario 4	55	7	3	169456
Large Scenario 1	90	12	2	737191
Large Scenario 2	90	12	2	737191
Large Scenario 3	90	12	2	737191
Large Scenario 4	90	12	2	737191

For the small network configurations, the number of actions is 54 or 55 depending on the number of randomly generated attacks as shown in Table 6.4. One action is added per each attack in a scenario to the basic 53 actions. Two or three attacks are randomly generated for each scenario because two actions are required to resolve certain attack types. This leads to two combinations for a scenario. One is three random attacks with a single action. The other is one random attack with two actions needed and one random attack with one action needed. Due to the different combinations of random attacks, the action set is slightly different but approximately the same.

The number of actions determine the size of the state-action tree. For example, with 54 actions the tree size is $54^3 + 54^2 + 54 + 1$ which equals 160435 nodes in the tree. Total nodes of a fully explored tree is listed in Table 6.4. This size is within the constraints previously mentioned and so we can fully explore this state-action tree.

The large network configurations have an increased number of hosts. Five hosts are duplicated from the small network configuration and given unique identifiers. The firewall host is not duplicated in order to have a central firewall host machine. As Table 6.4 shows, the action count and hosts are increased from the smaller configuration. The total number of nodes in the search space increases as well. The total number of nodes of the large network configuration makes the memory allocation work under the 8GB of RAM constraint.

6.6 Size Performance

There are several parameters that are important to the different aspects of performance. For the space performance, which also effects time performance, the number of actions from the network configuration has a significant effect on the size of the search space. As previously mentioned, the number of actions is the base of the exponential growth of the tree. Optimizing the number of actions gives better results in terms of search time and memory space needed. Figure 6.2 and Figure 6.3 show the small network configuration tree growth for all c values for the amount of time it takes $c = \sqrt{5}$ to get to 40,000 iterations.

This time varies for each scenario but generally stays around 200ms. These figures show the 95% confidence intervals for these averages at each point. To clearly see the plots of each *c* value, the appendix has each of these separated for better visualization.

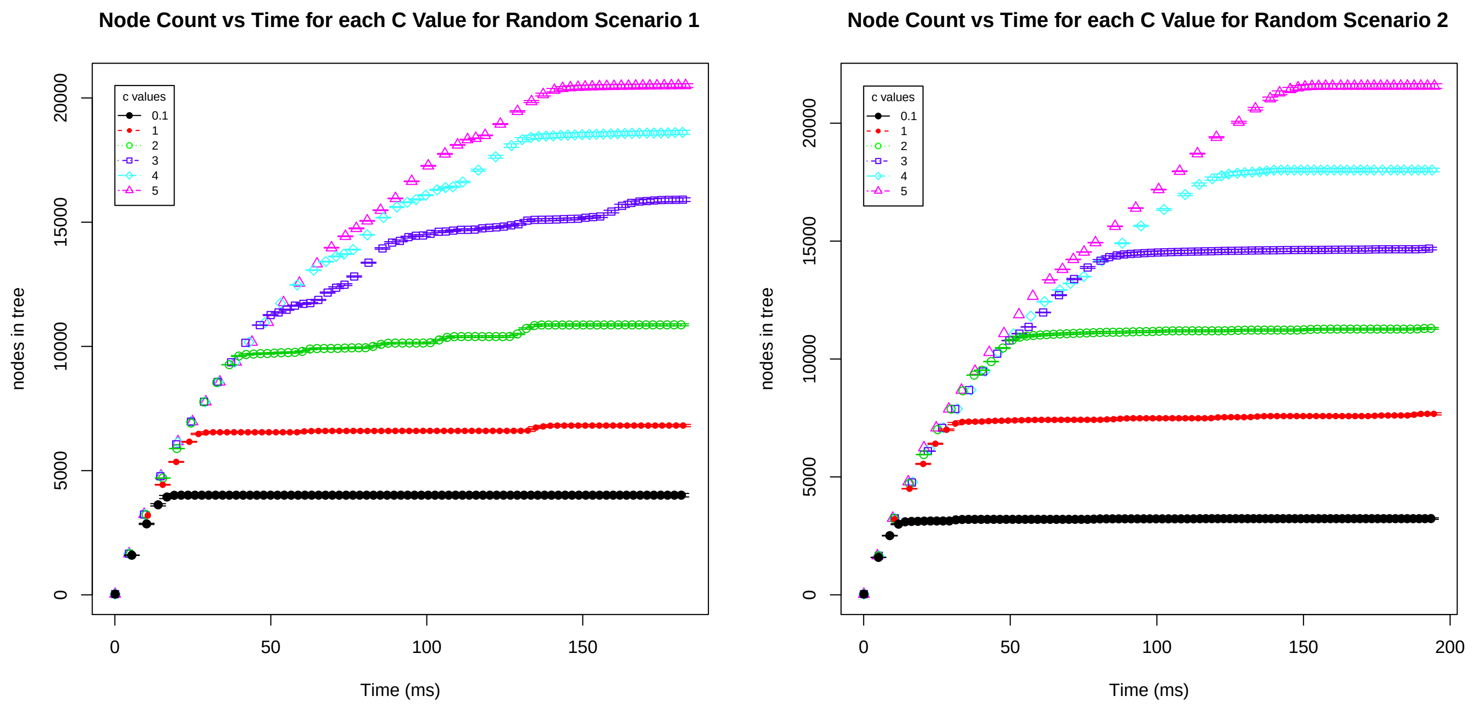


Figure 6.2: Node count per time with different *c* values for Random Scenario 1 & 2.

Each of these scenarios, additional actions are not created as it searches so growth in that regard is fixed. The first observation from these figures is that the *c* values influence the rate of the growth of the tree. A *c* value of $\sqrt{2}$ is given to be the optimal balance in [31] which the other *c* values are centered around. Two *c* values of $\sqrt{0.1}$ and $\sqrt{1}$ are the two below and $\sqrt{3}$, $\sqrt{4}$, and $\sqrt{5}$ are used above to observe the variations of changing this parameter. As Figure 6.2 shows, the values of $\sqrt{0.1}$ and $\sqrt{1}$ have very small growth once it plateaus. The smaller *c* values relate to the plateau height in the plots. However, a *c* value of $\sqrt{2}$ and higher demonstrate further growth once they have reach a plateau.

Small scenario 1 and 2 has similar growth patterns shown in Figure 6.2. The difference between them is that scenario 2 plateaus more quickly and has little growth afterward. As shown in these two scenarios, *c* controls the growth of the nodes in the search space. This growth slows down greatly around a certain threshold which depends on the *c* value used.

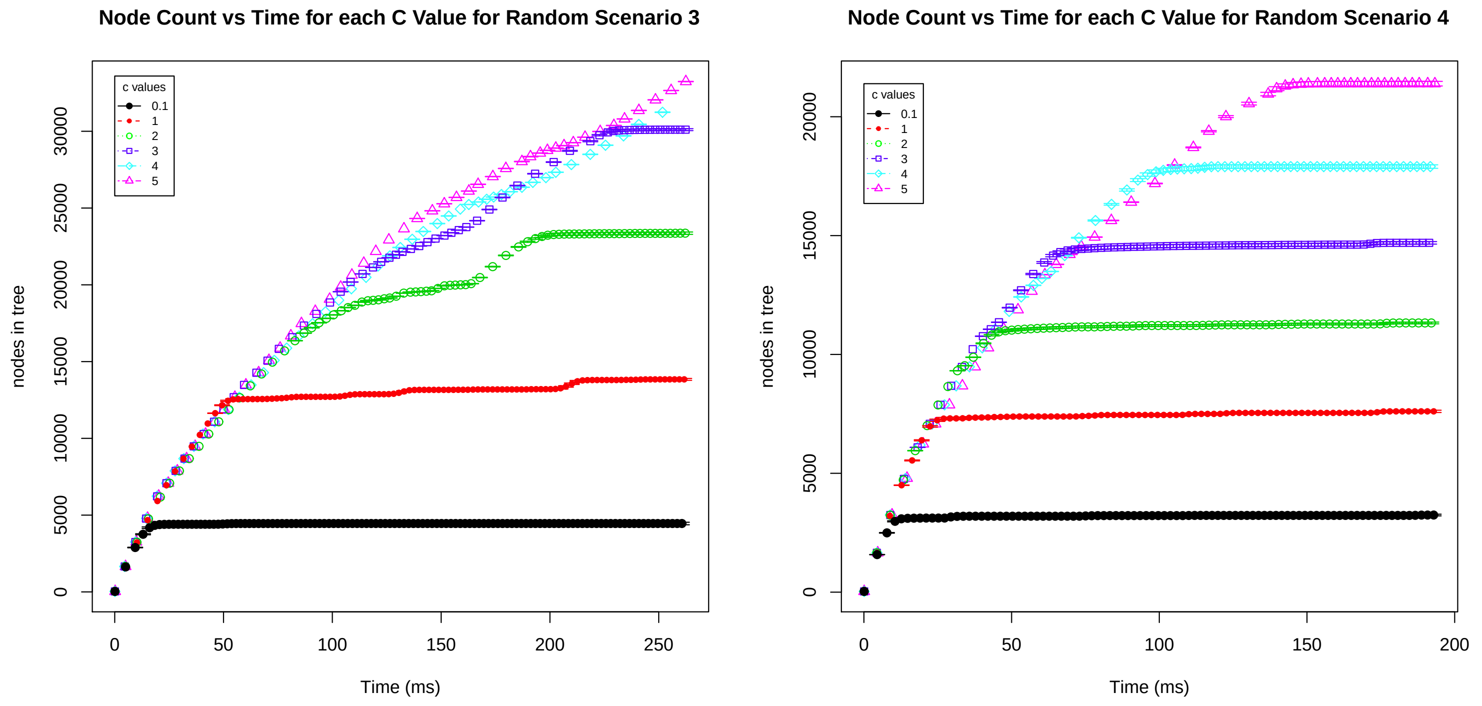


Figure 6.3: Node count per time with different *c* values for Random Scenario 3 & 4.

The last two small scenarios, 3 and 4, are shown in Figure 6.3. Small scenario 4 shows similar growth to the first two small scenarios, however, small scenario 3 has a different growth in the number of nodes. Seen in the same time frame, the growth does not settle as quickly as the other small scenarios did. However, a trend is seen as the *c* values approaches plateaus and slows down in growth except for $c = \sqrt{4}$ & $\sqrt{5}$. Small scenario 3 has a larger node growth compared to the other small scenarios. It reaches above 30,000 nodes while the other scenarios plateau at around 20,000 nodes. In some cases, a 10,000 node difference is observed in growth for these high *c* values when comparing these figures.

The total amount of nodes for a fully explored tree is shown in Table 6.4 for the smaller network configuration; these plots show that the growth is still much smaller than the max number of nodes. The node growth for these high *c* values is approximately 21% (35,000 / 160,000) of the overall possible search space which means the time to find the optimal plan is efficient and is always found for some *c* values.

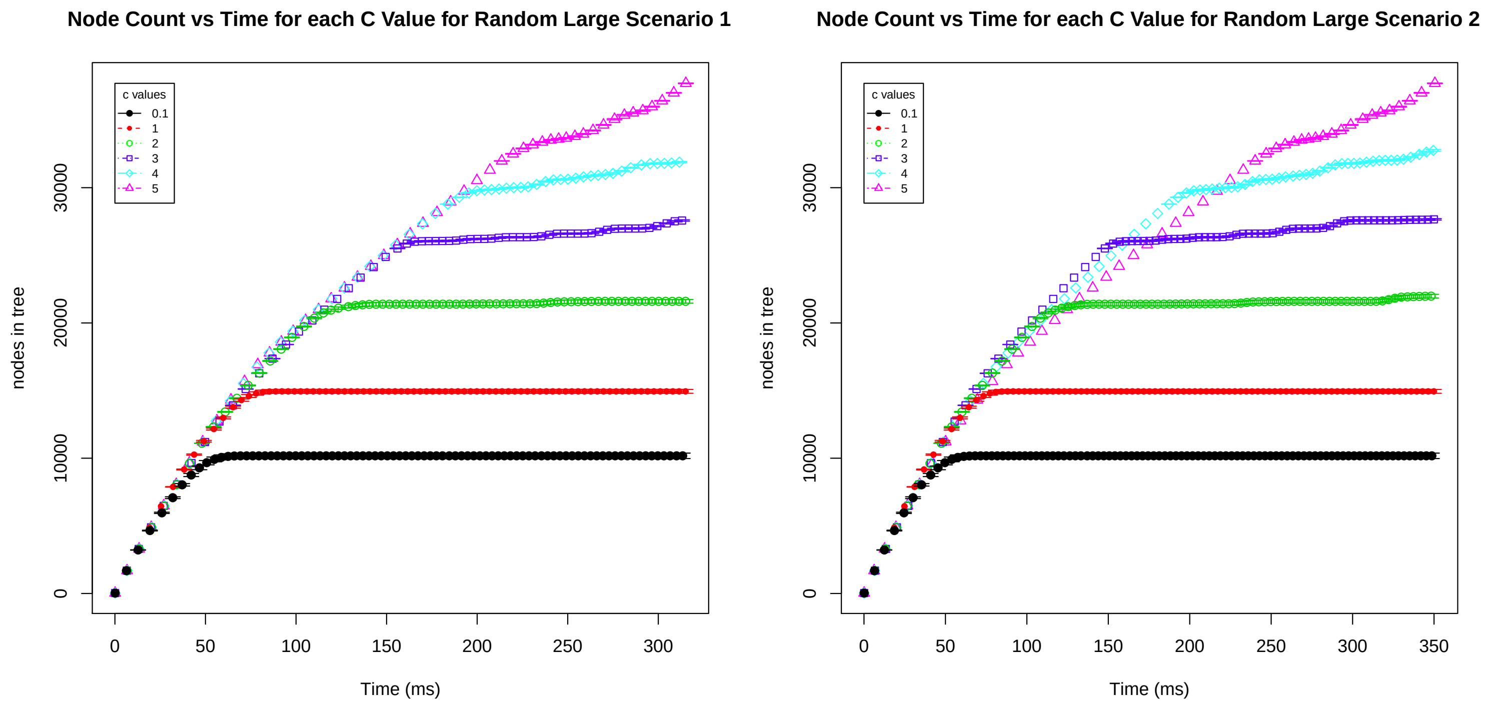


Figure 6.4: Node count per time with different *c* values for Large Random Scenario 1 & 2.

Large random scenarios results for node growth demonstrate more early plateaus except when using some of the higher *c* values. There is no significant variation between the large scenarios as there were in the small scenarios. The node count is on the same level as in the third small random scenario. The figures indicate that increasing the action count does not completely control node growth during the searching. However, the depth of the searching, the *c* parameter, and the number of iterations performed have a impact on the node growth. High *c* values and larger depths causes the search tree to explore more

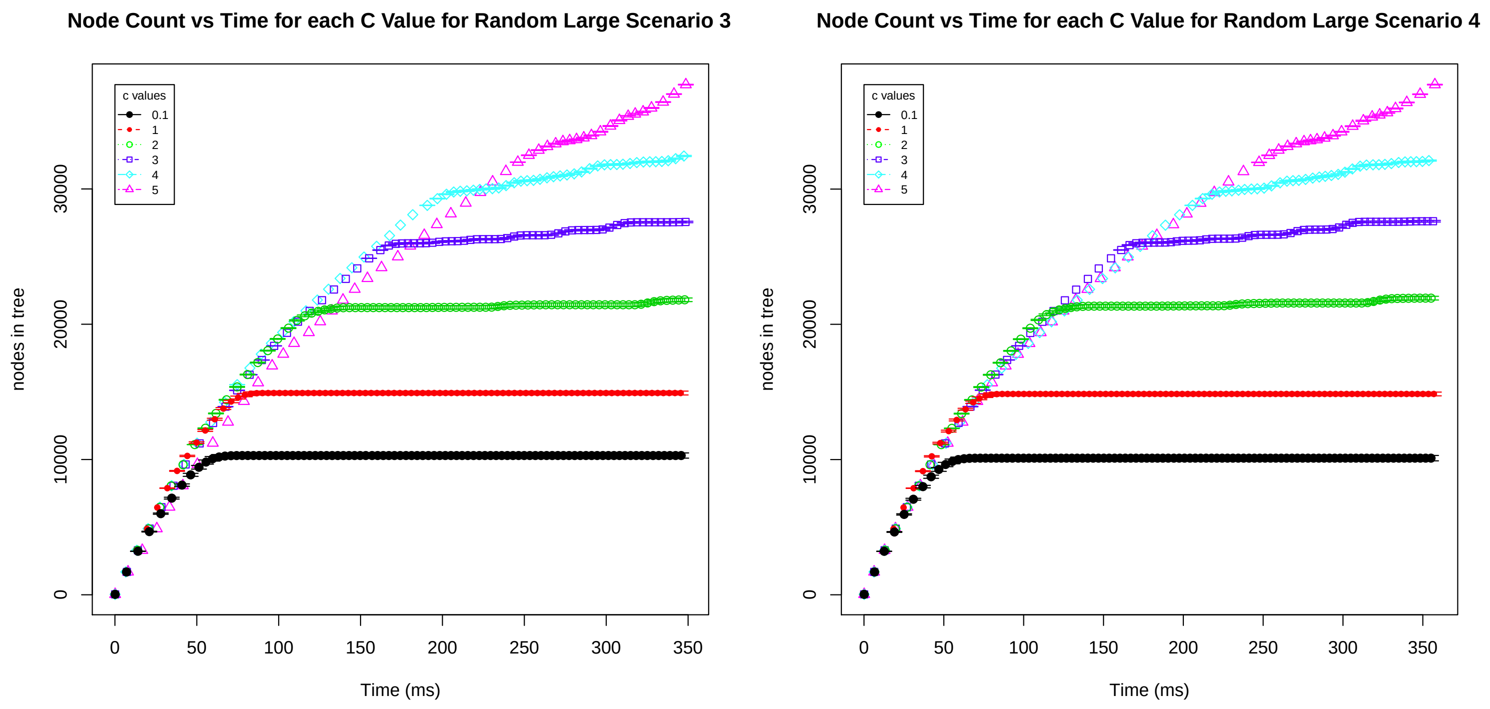


Figure 6.5: Node count per time with different *c* values for Random Scenario 3 & 4.

over each iteration. Increasing the action set cardinality also means less of the tree will be search using the same parameters as used with a lower cardinality. Thus, the optimal plan might not be found quickly.

In all the scenarios, plateaus of node growth are observed to be dependent on the *c* value. If memory space is a constraint in the system environment, decreasing the value of *c* is one method to limit growth of the search tree. There is a tradeoff in using this method that the optimal plan may not be found for some *c* values less than $\sqrt{2}$.

6.7 Run-time Performance

The important aspect in choosing to use the UCT algorithm is for it to perform in a real-time environment. Run time data for one small network configuration is illustrated in Figure 6.6 and one large configuration is shown in Figure 6.7. These figures show the time in the milliseconds that 10 iterations took to performed. Although the plots appear to be cluttered, they give an overall visual comparison between the *c* values. All small random scenarios generally had an average time of 0.02ms for 10 iterations. Similar plots for all the scenarios can be found in Appendix E.

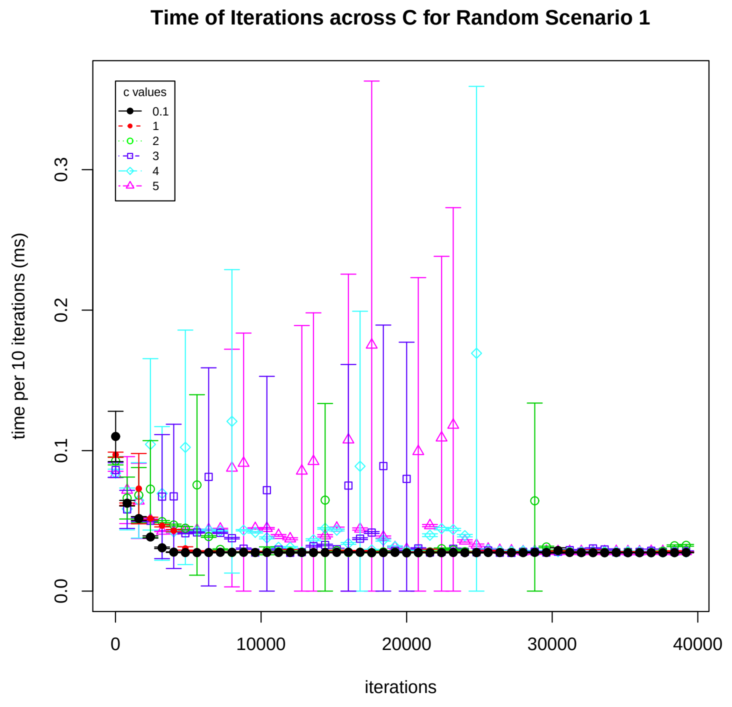


Figure 6.6: Average Time for 10 iterations with different *c* values for Random Scenario 1.

All of these figures have noticeable spikes in time which at first may seem to be random. These spikes are caused by Java optimizing the space; this is done for some data structures such as ArrayLists which are used in this program. These spikes occur frequently in the early iterations due to the growth of the search tree. The rate of growth is initially large and starts to plateau depending on the *c* value. Java does optimizing more in the earlier iterations, but because the focus is real-time performance, the beginning portion of run-time is important to observe. In addition, we would not expect the UCT algorithm to run for a long duration in a real world cyber defense application.

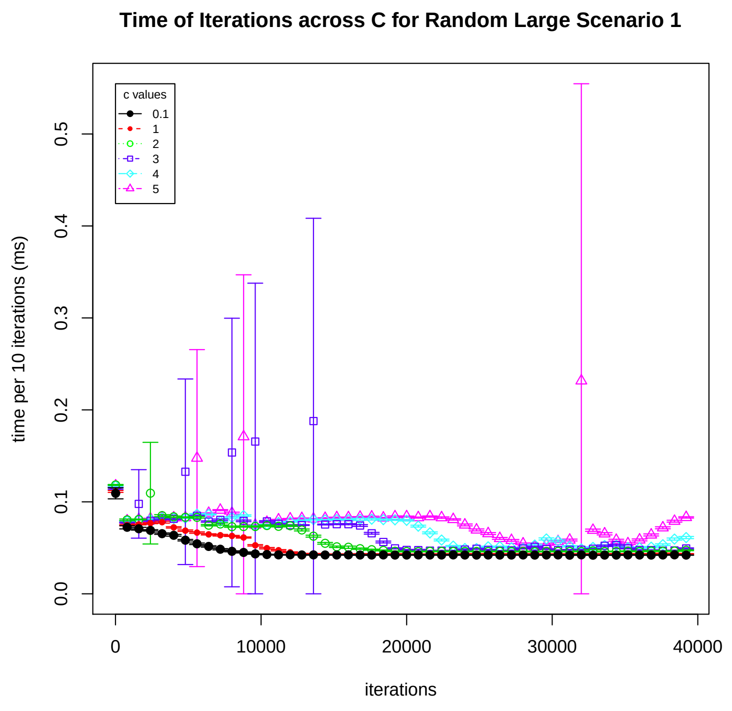


Figure 6.7: Average Time for 10 iterations with different *c* values for Large Random Scenario 1.

The time per 10 iterations goes up slightly for the large scenarios shown in Figure E.5 and Figure E.7. The time spikes are apparent in the large scenarios as well. As the iterations are processed, the time per iteration decreases as growth stabilizes. Similar to the small scenario figures, *c* causes the time to take longer to settle. This is due to the higher *c* values taking longer to reach their node growth plateaus. Once there is minor growth between the iterations, the program runs faster due to less optimizing of the allocated memory space. The large random scenarios have a larger time per 10 iterations. This is noticeable when comparing them against the smaller scenario figures. Depending on the *c* value, the time per 10 iterations for the large random scenarios is approximately 0.5ms to 1ms. Some *c* values for the large scenarios show no early signs of converging to a steady time per iteration. This is due to the tree still growing rapidly and not reaching a steady plateau state.

UCT finding the optimal plan quickly is an important part of the research because of the real-time requirement. Figure 6.8 explains the average time to find the optimal plan at a 95% confidence interval. The results for the small scenarios show that as the c value is increased, the longer it takes to find the optimal plan. This is expected due to increasing the c value results in more exploring which in turn takes longer to find the optimal plan. Small scenario 1 appears to be the only scenario that is different amongst the small scenarios. The other small scenarios illustrate an expected increase in average time as the c value increases. Figure 6.8 indicates that $c = \sqrt{1}$ is unusually large compared to other small scenarios average time values. The small scenario results show quick performance given the time allotted to find the optimal plan. Large scenarios illustrates the trend in average time to find the optimal plan across c values more clearly.

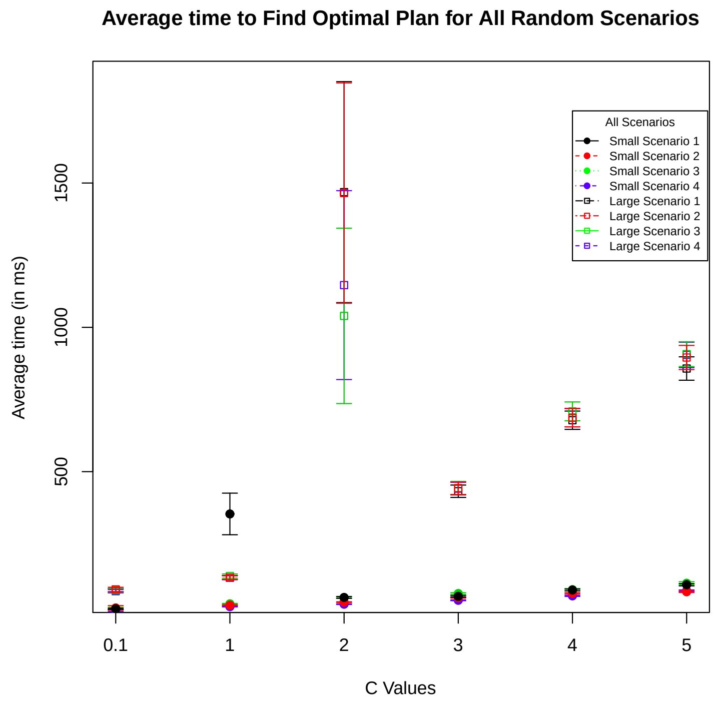


Figure 6.8: Average time to find optimal plan (in ms) for all scenarios.

All large scenarios have unexpected large time averages for $c = \sqrt{2}$. This is unexpected due to the value $c = \sqrt{2}$ being reported as the most balanced value. The large time average for large scenario 1 and 2 is likely correct due to the two scenarios being essentially the same. This occurrence means there were 2,000 runs of the same scenario which resulted in the same average. Multiple runs in those scenarios resulted in long times to find the optimal plan. This can be a result of the growth of lower c values becoming negligible after a certain number of iterations. This slow down in growth leads to finding the optimal plan much later. Even though the c values smaller than $c = \sqrt{2}$ have a lower reported average time, they did not find the optimal plan every time. In some cases, the c value used causes it to not be found even half of the time.

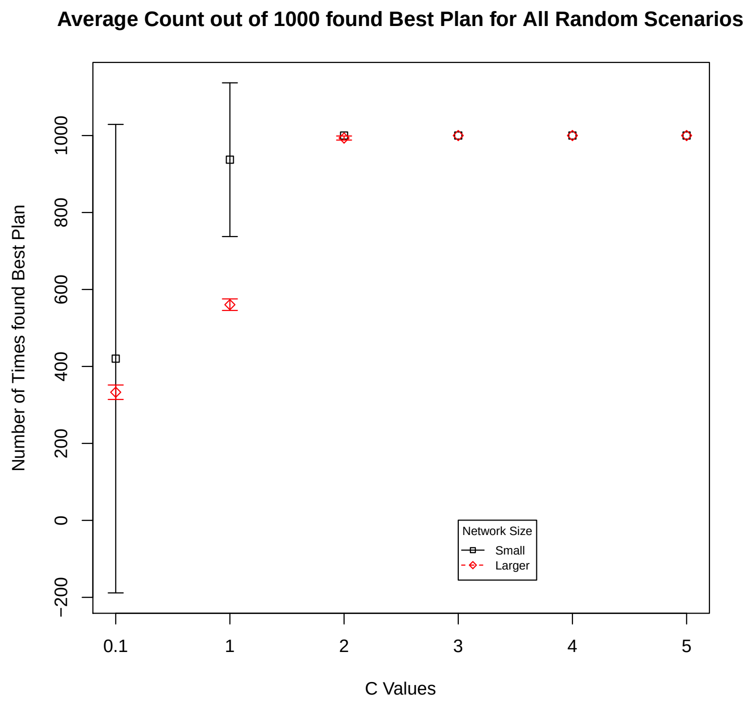


Figure 6.9: Number of times found optimal plan out of 1000 for all scenarios.

As presented in Figure 6.9, the average count out of 1,000 runs for the small network scenarios for finding the optimal plan contribute to the difference seen in Figure 6.8.

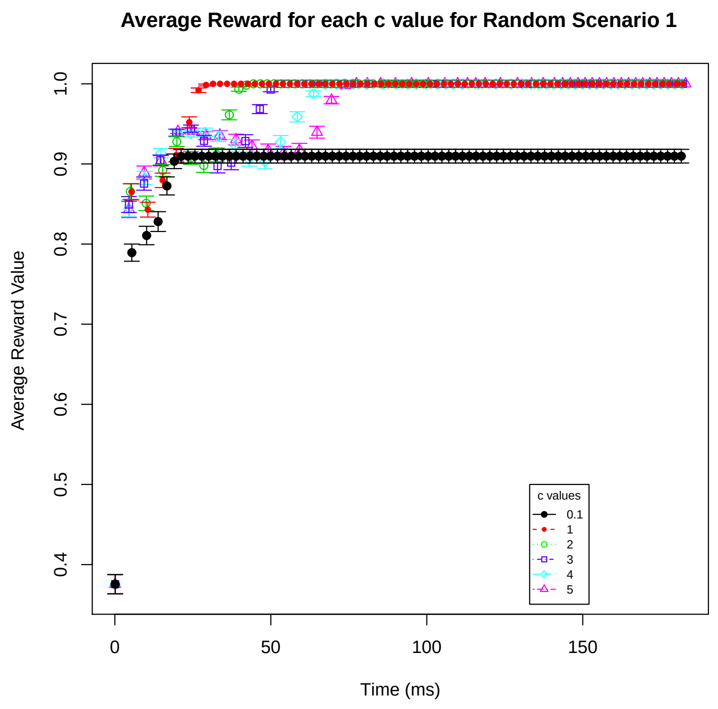
Figure 6.9 shows that $c = \sqrt{0.1}$ and $c = \sqrt{1}$ do not always find the optimal plan. Due to this, those c values have a lower number of runs that found the optimal plan when averaging the results. Since these two c values reach their growth limitations more quickly, they are not likely to find the optimal plan at later times as with other higher c values.

It is clear from Figure 6.9 that for all network configurations a c value of $\sqrt{2}$ or greater is needed to find the optimal plan with greater frequency. The balanced, $c = \sqrt{2}$ value appears to not find the optimal plan in all of the runs for the large scenarios. It is slightly below 1,000 times the optimal plan ideally should be found. The large random scenarios average times found per c value trend is similar to the small scenarios. As c value increases, the number of times the optimal plan is found increases, and this is expected due to more exploring.

6.8 Reward Performance

Having real-time performance is not the only important parameter in the NDPS to evaluate. The reward for a given plan generated from the UCT algorithm must also be considered. The average reward is calculated by using the last reward value after implementing the plan which consists of three actions. The reward is normalized as presented in the previous chapter so that implementing an optimal plan gives a value of one. In the following four small random scenarios, Figure 6.10 and Figure 6.11 show the results of the average reward over time for c values tested. The individual c value plots can be seen in the appendix for greater clarity.

One observation apparent from all the small scenario plots is that the value $c = \sqrt{0.1}$ is lower on average for small scenarios 1, 2, and 4. This is due to how smaller c values exploit rather than explore. After a certain amount of growth, it no longer explores as much as the other c values.



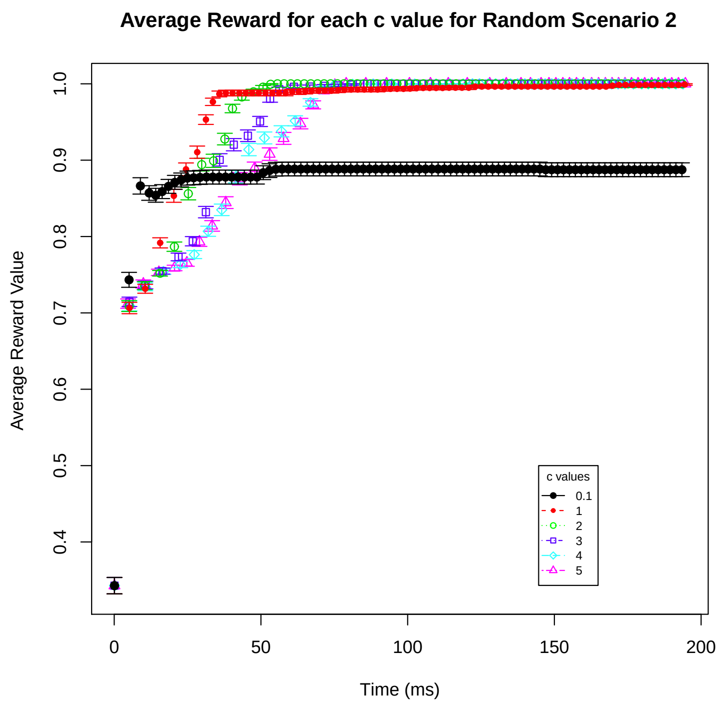
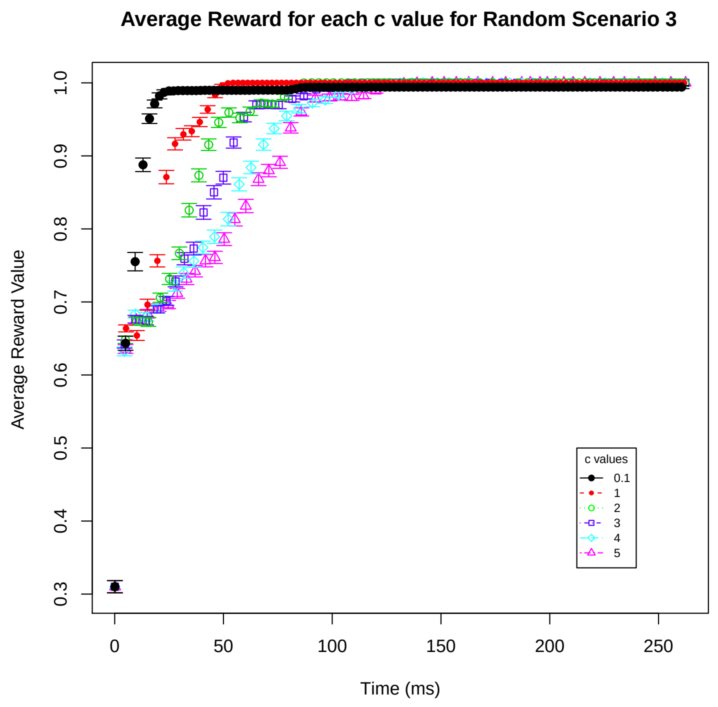


Figure 6.10: Average Reward vs Time with different *c* values for Random Scenario 1 & 2.



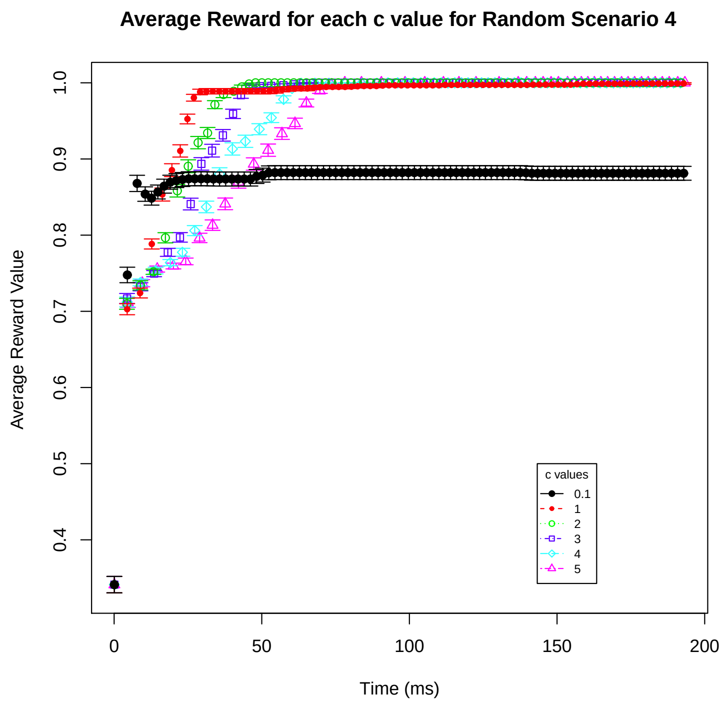


Figure 6.11: Average Time vs Time with different *c* values for Random Scenario 3 & 4.

All scenarios indicate how increasing the c value causes the average reward to be found at later times due to more exploring. A value of $\sqrt{0.1}$ can on average achieve the reward faster, however, it can also reach a state where it does not find the optimal plan. This appears in Figure 6.10 and Figure 6.11, the average reward is noticeable lower and shows a significant increase in time consumed. While many of the figures show an increasing reward over time, in some cases such as Figure 6.10, there can also be minor decreases in the average reward.

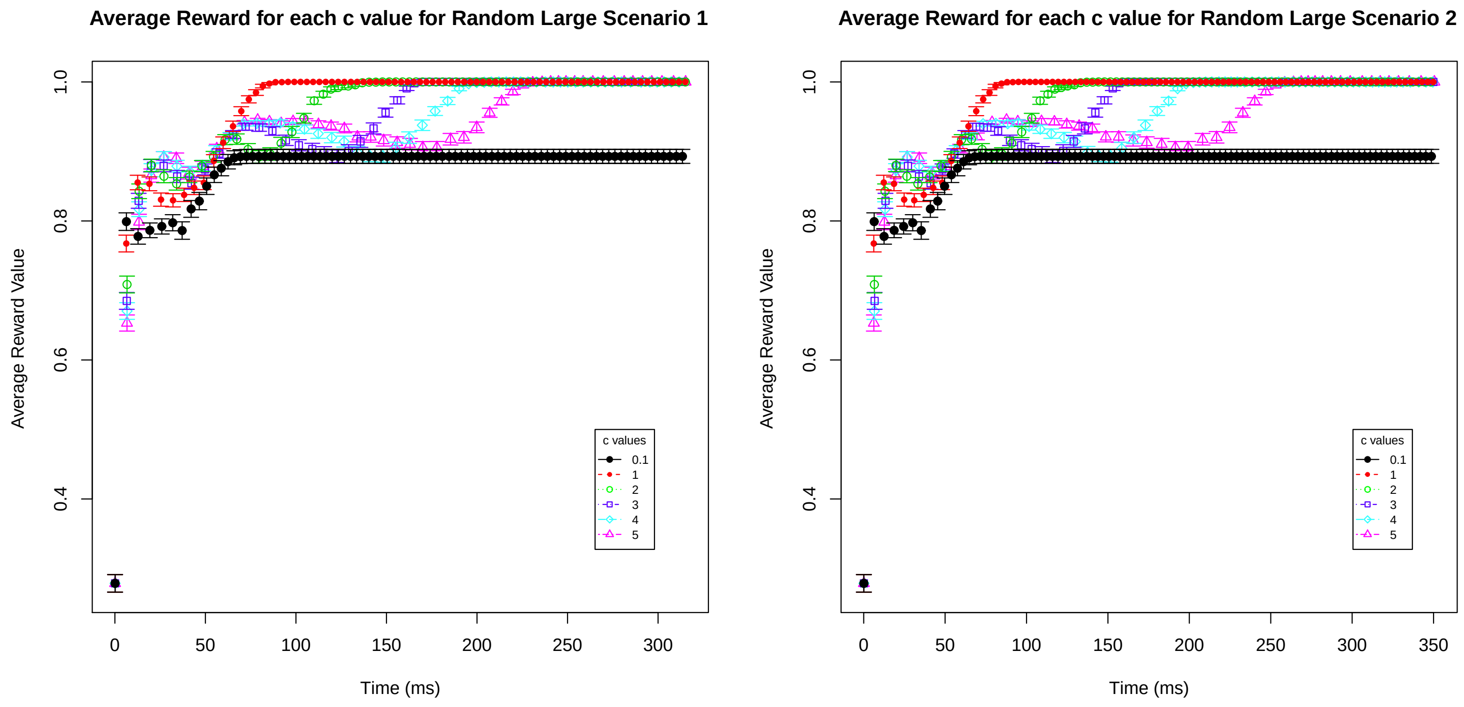


Figure 6.12: Average Time vs Time with different c values for Large Random Scenario 1 & 2.

The decreases in average rewards are seen in Figure 6.12 as well. These scenarios have an average reward increase overall but at times experience a drop in average reward as it climbs to the max reward. It is likely the decreases in average rewards are caused by types of attacks which need two optimal actions to fully resolve. For instance, a plan with less reward value such as “shutdown host” is found to have the best average reward until both of the two optimal actions are found. As it is exploring, the two optimal actions areas in the tree are found that have better Q values. The algorithm operates on Q values which are based on the expected reward. Overall, reward values have an increasing trend except if stopped at certain times. Then the reward may be slightly less since there was not enough time to explore the two levels needed for both optimal actions.

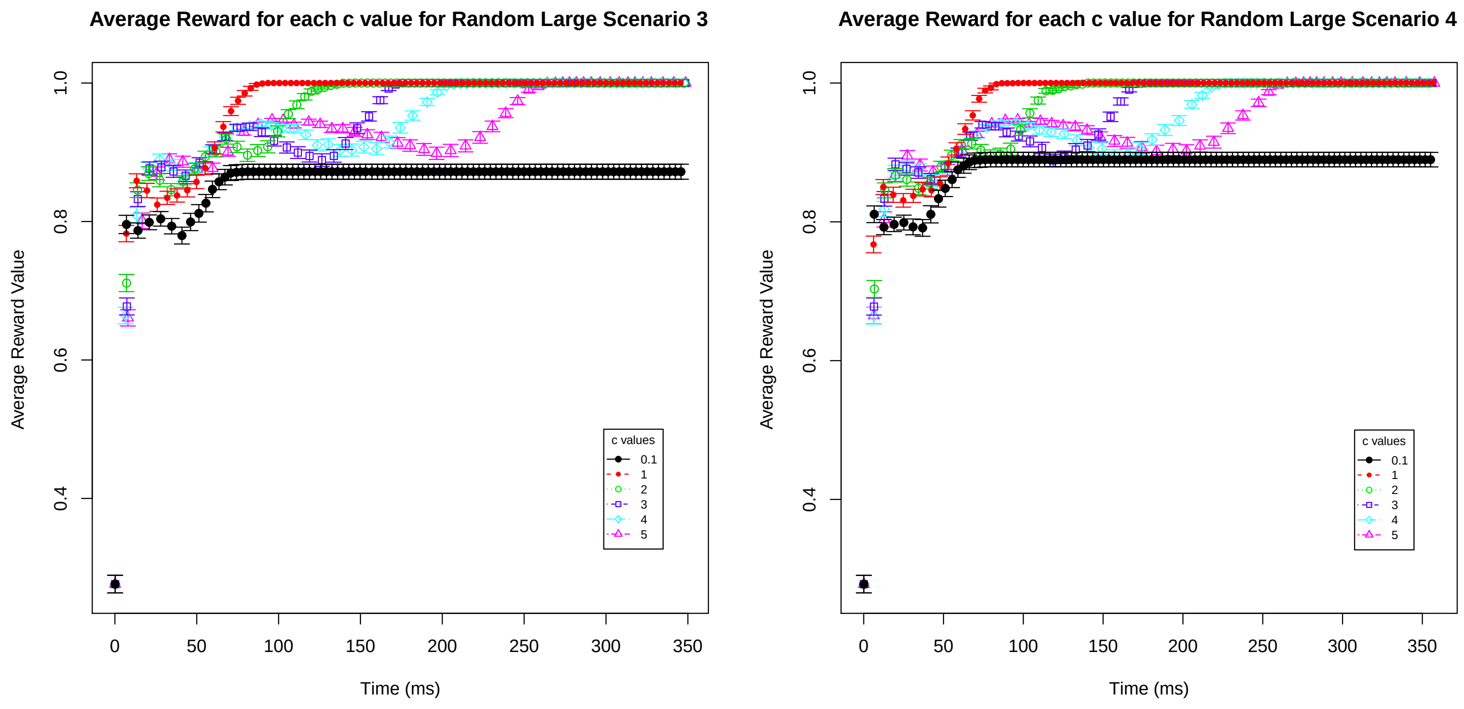


Figure 6.13: Average Time vs Time with different c values for Large Random Scenario 3 & 4.

There are some data inconsistencies that appear contradictory. In Figure 6.8, the large scenarios took over one second for c value of $\sqrt{2}$ to find the optimal plan. However, Figure 6.13 shows the average reward value at one before one second elapses for all c

values. This is due to the action sequence of the plan. For instance in all large random scenarios, the optimal plan had a reward sequence 0.5 $\rightarrow$ 0.95 $\rightarrow$ 1.0. The system found the sequence 0.45 $\rightarrow$ 0.95 $\rightarrow$ 1.0 first. This is not optimal since it does not repair the network in most efficient way according to the reward calculations. Overall, the plan to repair the network is found early but not in the optimal sequence.

6.9 Scalability in Network Size

Another aspect to investigate in using the UCT algorithm to find the optimal plan is discovering if it is suitable for larger network sizes. The two network sizes utilized are due to the constraints of using a computer with 8GB of memory and exploring the state-action tree fully, which ideally would defeat the purpose of using UCT. However, it is necessary to fully explore to validate what the optimal plans are in the state-action tree for this work. This leaves the question of whether UCT algorithm can scale to larger network sizes.

There are at least two optimizations that could be made to accommodate larger networks, altering action sets to solely relevant actions and optimizing by packaging actions. The action set used in these scenarios contained several actions that were not relevant to changing the state of the network to resolve the attack events. By executing only relevant actions to resolve network attacks, the action set can be reduced. The other optimization is the packaging of actions which would reduce the number of searchable actions required. For instance, to resolve a FTP Brute force type of attack, two optimal actions are necessary in this system. These two actions could be packaged together so that when “Block IP” is found it runs additional actions as needed for this attack type. Thus, the reduction in searchable actions for a network state would extend to larger networks better.

Upon analyzing the results from the configuration used in Figure 6.8, the optimal plan is mostly found within one second. Depending on the c value, larger network sizes than what is used in these simulations can also achieve real-time performance. Without testing more network sizes, it is not possible to know the exact limit of when real-time performance ceases. With around 12 hosts and 90 actions, the performance would work in a real-time environment. The trend in Figure 6.8 for large scenarios' average time appears to be growing non-linearly as the *c* value increases. This is slightly noticeable in the smaller scenarios' trend across *c* values as well. This trend may indicate that it would not be a linear growth in average time as the network size increases. As a result, using one central automated planner would not likely be the solution for enterprise level of networks. Constructing parallel planning agents can be a solution to resolve scalability issues.

VII. Conclusions

Real-time classification, learning rewards, and UCT planning methods are important areas to investigate in creating a cyber defense system. The components established for simulations were tested individually and proved that their performance is able to operate in real-time. Unfortunately, due lack of computer memory and time constraints not all the components are not tested together to analyze overall combined performance. To resolve these issues, a comprehensive system with all components could be built and run on a computer that eliminates space constraints to perform further testing. The results demonstrates there is potential to use the classifiers and planners together for a cyber defense system. Chapter III demonstrates that abstracting and classifying sample data can increase run time performance without significant loss of accuracy. Two widely-used classifiers are also compared to determine the best performance of a real-time application. The classifier data set used is old, therefore future work could be to obtain a more current data set to train for recent types of attacks. Ideally, the same methods could be used on new data sets but further research would be needed to see if any significant problems arise by using a newer data set.

The NDPS demonstrates flexibility and real-time application in cyber defense. More scalability research could offer a better indication of how greater search depths and action sizes effect the performance of the UCT algorithm. Additionally, more complex randomized scenarios could be created to see how the NDPS handles them. The reward calculations have several parameters involved and can become convoluted without proper attention. Defining rewards for states is difficult in the vast majority of domains, and capturing the significance of what is important is a major issue. The neural network research helps alleviate capturing what the reward model of the network operator does and could be extended to domains other than cyber. Although the neural network does have issues of having to be retrained with any significant changes to the model and is susceptible to some noise in collection of the samples, there are several other future work areas that could be explored.

One goal was to research plan effectiveness. Once a plan was created and implemented, feedback would be given to NDPS which allowed it to update the plan effectiveness for the previously generated plan. This work was simulation but built in a way to easily translate to future emulation work using a created virtual network that captures sensor data in one XML file. Having the ability to learn plan effectiveness could potentially enable the system to adapt to the constant changing environment in the cyber domain. Another goal would be if the scalability of UCT did not work as expected for real-time applications, then there is possible application for parallel UCT agents. Having UCT agents responsible for certain parts of the network, thus subsets of the actions, would allow faster processing. A possible complication would be if those actions are dependent across the agents. If so, communication between them would be needed to ensure proper plan building. There are plenty of additional avenues this research could be directed, and the work presented offers results that demonstrate it can operate within the cyber domain's real-time requirement.

Appendix A: Additional Classifier Data

Table A.1: Confusion matrix for abstracted 14 classes with K-Means.

classified as →	a	b	c	d	e	f	g	h	i	j	k	l	m	n
a = normal	35627	2	8535	2964	1397	555	74	47	6837	1739	358	10	3658	312
b = neptune	0	21569	1	0	230	0	1549	6090	0	3	0	386	0	0
c = warezclient	0	0	807	0	0	0	1	0	1	2	0	0	47	19
d = ipsweep	0	0	14	3113	66	5	0	338	14	16	0	2	8	3
e = portsweep	1	105	6	5	1878	194	10	688	6	6	0	6	0	0
f = teardrop	0	0	0	72	540	162	0	49	58	11	0	0	0	0
g = nmap	0	99	0	1113	50	29	157	0	35	0	0	0	0	0
h = satan	2	76	31	89	568	166	0	2174	159	213	0	1	9	2
i = smurf	0	0	0	722	64	63	0	0	1538	259	0	0	0	0
j = pod	0	0	0	87	6	11	0	0	64	33	0	0	0	0
k = back	908	0	0	0	0	0	0	0	0	0	47	0	1	0
l = guess_passwd	0	0	0	42	3	0	1	0	0	0	0	5	1	0
m = buffer_overflow	0	0	17	0	0	0	0	0	0	0	2	0	11	0
n = warezmaster	0	0	14	6	0	0	0	0	0	0	0	0	0	0

Table A.2: Summary statistics of 14 classes with K-Means.

Summary Statistics

Correctly Classified Instances	67121 (53.3125 %)
Incorrectly Classified Instances	41953 (33.3222 %)
Kappa statistic	0.4837
Mean absolute error	0.0549
Root mean squared error	0.2344
Relative absolute error	73.6778 %
Root relative squared error	121.5215 %
UnClassified Instances	16827 (13.3653 %)
Total Number of Instances	125901

Table A.3: Confusion matrix for abstracted 14 classes with K-Means.

classified as →	a	b	c	d	e	f	g	h	i	j	k	l	m	n
a = normal	36883	18	6640	3604	1421	117	101	627	8197	1553	4501	371	2013	200
b = neptune	0	13153	0	0	1216	0	5772	4861	21	115	0	935	1	2
c = warezclient	0	0	740	13	0	0	1	1	0	2	45	19	38	31
d = ipsweep	0	0	11	3090	93	0	0	282	45	21	16	0	4	0
e = portsweep	1	68	2	5	1838	12	32	784	84	3	3	5	0	1
f = teardrop	0	0	0	267	488	9	0	70	57	1	0	0	0	0
g = nmap	2	162	0	1134	44	0	92	2	46	0	0	0	0	0
h = satan	7	54	50	181	984	52	15	1778	113	134	9	10	6	2
i = smurf	115	0	44	1095	159	0	0	44	995	194	0	0	0	0
j = pod	0	0	7	123	25	0	0	9	10	27	0	0	0	0
k = back	866	0	0	0	0	0	0	0	0	0	90	0	0	0
l = guess_passwd	0	0	0	40	1	0	0	0	0	0	1	10	0	0
m = buffer_overflow	0	0	10	2	0	0	0	0	0	0	11	0	7	0
n = warezmaster	0	0	8	12	0	0	0	0	0	0	0	0	0	0

Table A.4: Summary statistics of abstracted 14 classes with K-Means.

Summary Statistics

Correctly Classified Instances	58712 (46.6335 %)
Incorrectly Classified Instances	50574 (40.1697 %)
Kappa statistic	0.3829
Mean absolute error	0.0661
Root mean squared error	0.2571
Relative absolute error	89.6891 %
Root relative squared error	135.0116 %
UnClassified Instances	16615 (13.1969 %)
Total Number of Instances	125901

Table A.5: Confusion matrix for 14 classes with C4.5.

classified as →	a	b	c	d	e	f	g	h	i	j	k	l	m	n
a = normal	66819	9	127	150	23	18	51	42	48	14	36	3	0	3
b = neptune	6	41153	0	2	22	0	0	31	0	0	0	0	0	0
c = warezclient	124	0	763	1	0	0	0	0	0	0	1	0	0	1
d = ipsweep	74	0	0	3415	0	1	99	0	0	10	0	0	0	0
e = portsweep	13	51	0	3	2851	0	0	13	0	0	0	0	0	0
f = teardrop	1	0	0	0	0	830	6	55	0	0	0	0	0	0
g = nmap	48	0	0	824	0	12	603	4	0	2	0	0	0	0
h = satan	144	20	0	7	11	68	6	3375	1	1	0	0	0	0
i = smurf	105	0	0	5	0	0	0	0	2506	30	0	0	0	0
j = pod	7	0	0	5	0	0	0	0	64	125	0	0	0	0
k = back	795	0	0	0	0	0	0	0	0	0	161	0	0	0
l = guess_passwd	4	0	0	0	0	0	0	0	0	0	0	49	0	0
m = buffer_overflow	26	1	2	0	0	1	0	0	0	0	0	0	0	0
n = warezmaster	3	0	2	0	0	0	0	0	0	0	0	0	0	15

Table A.6: Summary statistics of 14 classes with C4.5.

Summary Statistics	
Correctly Classified Instances	122665 (97.4297 %)
Incorrectly Classified Instances	3236 (2.5703 %)
Kappa statistic	0.9572
Mean absolute error	0.0054
Root mean squared error	0.0532
Relative absolute error	6.2446 %
Root relative squared error	25.6353 %
Total Number of Instances	125901

Table A.7: Confusion matrix for abstracted 14 classes with C4.5.

classified as →	a	b	c	d	e	f	g	h	i	j	k	l	m	n
a = normal	66698	32	0	261	76	11	2	68	189	1	4	1	0	0
b = neptune	3	40819	0	81	5	0	0	306	0	0	0	0	0	0
c = warezclient	888	0	0	1	0	0	0	0	0	0	1	0	0	0
d = ipsweep	62	0	0	3510	0	0	0	0	27	0	0	0	0	0
e = portsweep	10	158	0	64	2578	0	0	120	1	0	0	0	0	0
f = teardrop	295	0	0	0	0	474	0	123	0	0	0	0	0	0
g = nmap	209	0	0	973	0	31	265	9	6	0	0	0	0	0
h = satan	408	180	0	6	8	440	0	2582	9	0	0	0	0	0
i = smurf	0	0	0	0	0	0	0	0	2646	0	0	0	0	0
j = pod	2	0	0	0	0	0	0	0	198	1	0	0	0	0
k = back	866	0	0	0	0	0	0	0	0	0	90	0	0	0
l = guess_passwd	4	0	0	0	4	0	0	0	0	0	0	45	0	0
m = buffer_overflow	29	1	0	0	0	0	0	0	0	0	0	0	0	0
n = warezmaster	20	0	0	0	0	0	0	0	0	0	0	0	0	0

Table A.8: Summary statistics of abstracted 14 classes with C4.5.

Summary Statistics	
Correctly Classified Instances	119708 (95.0811 %)
Incorrectly Classified Instances	6193 (4.9189 %)
Kappa statistic	0.9173
Mean absolute error	0.0111
Root mean squared error	0.0748
Relative absolute error	12.8595 %
Root relative squared error	36.02 %
Total Number of Instances	125901

Appendix B: Average Times for Optimal Plan Plots

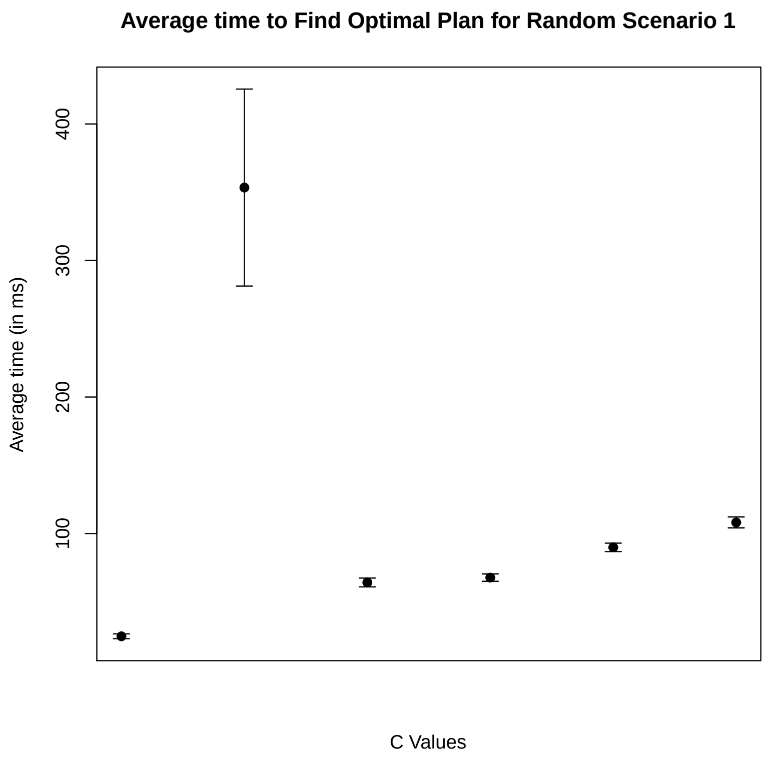


Figure B.1: Average time to find optimal plan (in ms) for Random Scenario 1.

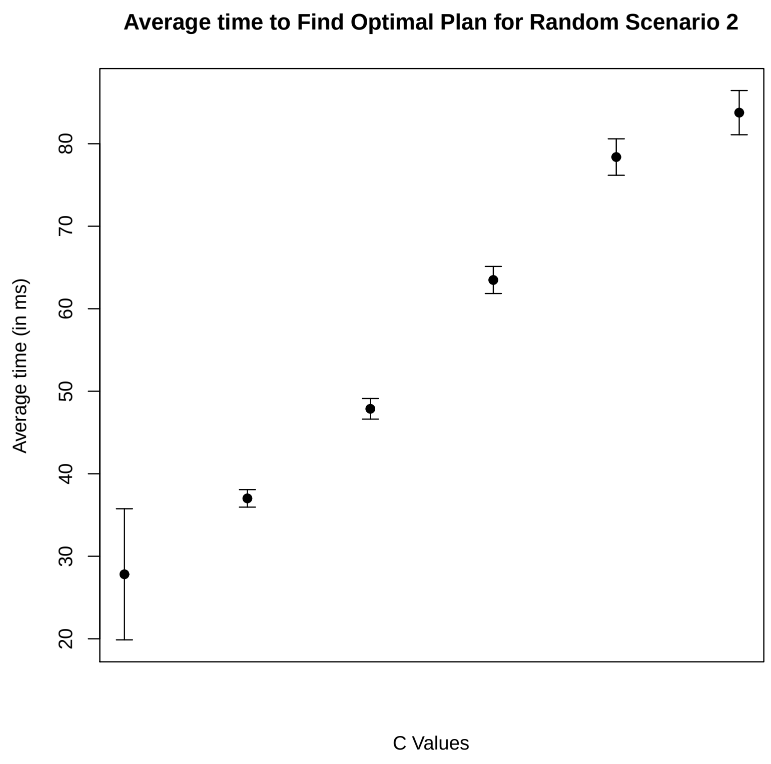


Figure B.2: Average time to find optimal plan (in ms) for Random Scenario 2.

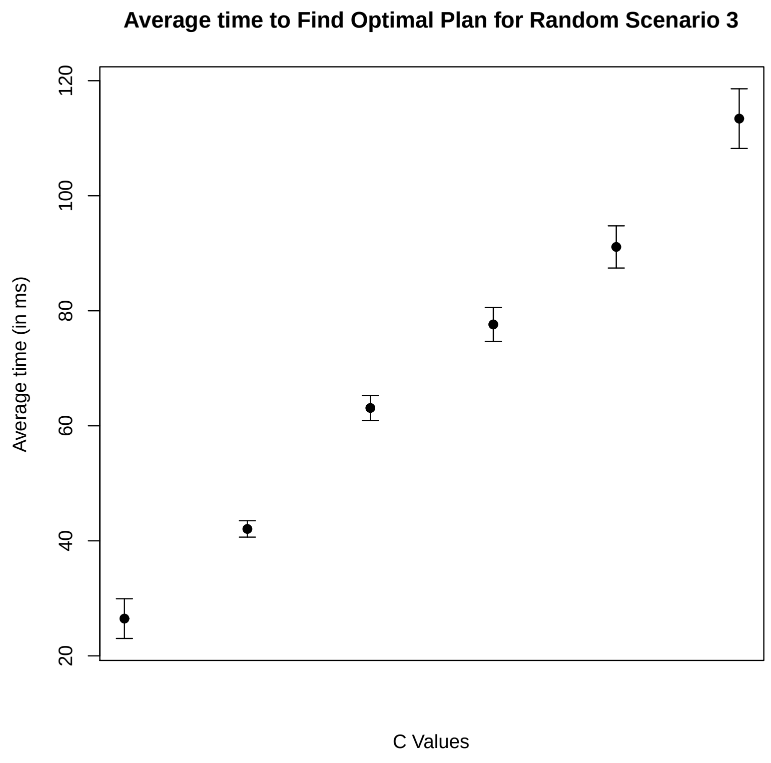


Figure B.3: Average time to find optimal plan (in ms) for Random Scenario 3.

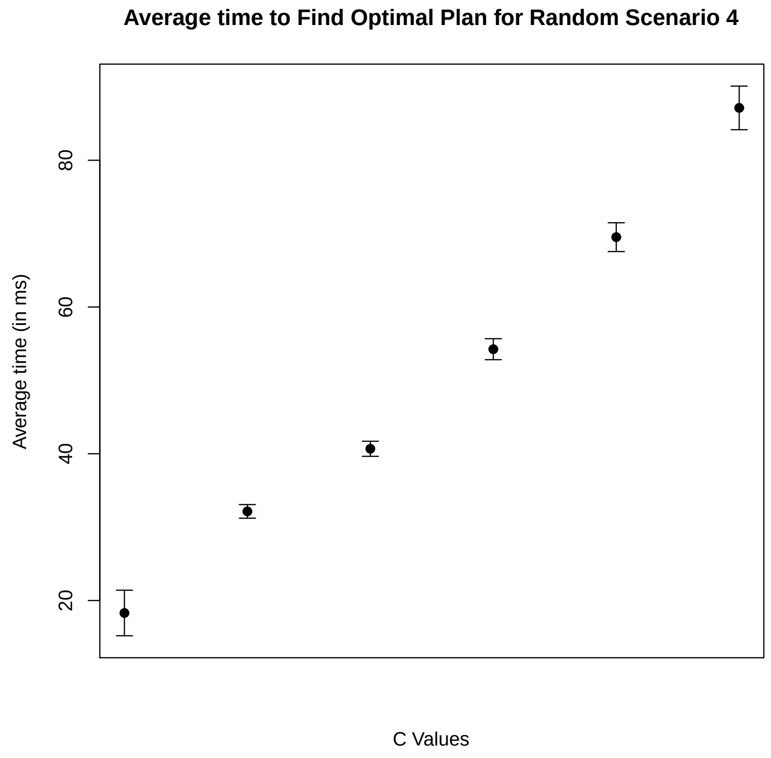


Figure B.4: Average time to find optimal plan (in ms) for Random Scenario 4.

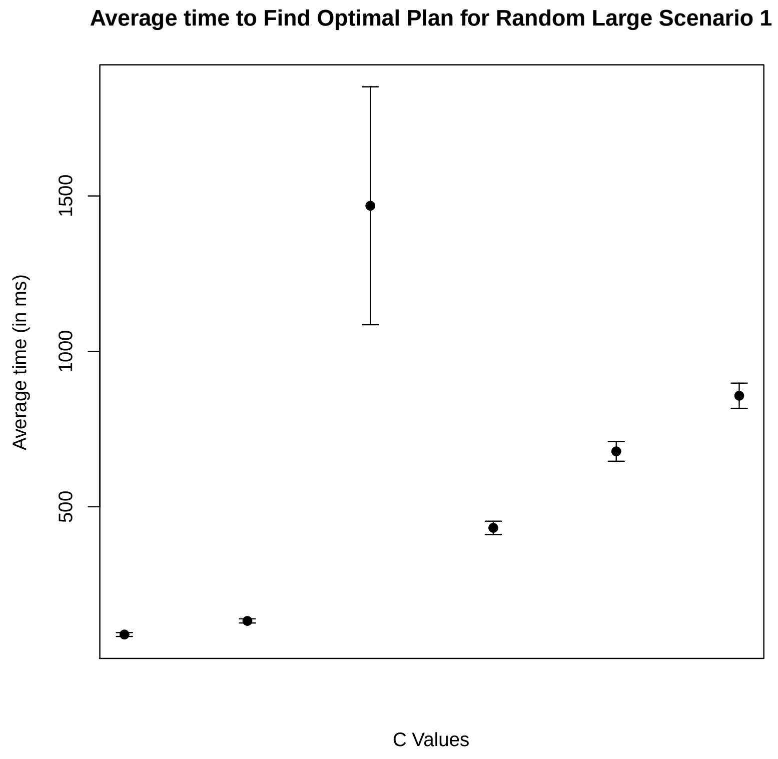


Figure B.5: Average time to find optimal plan (in ms) for Large Random Scenario 1.

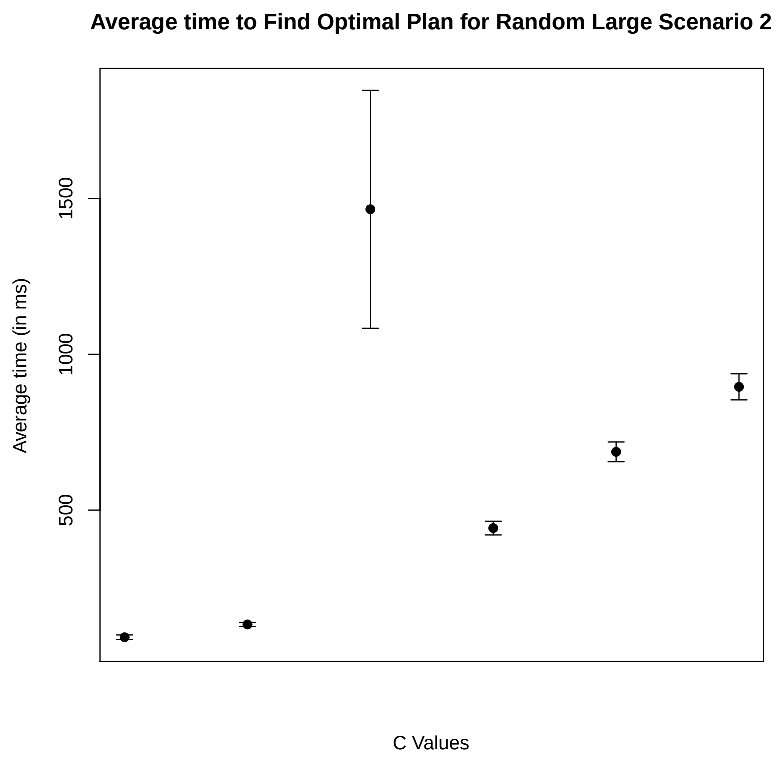


Figure B.6: Average time to find optimal plan (in ms) for Large Random Scenario 2.

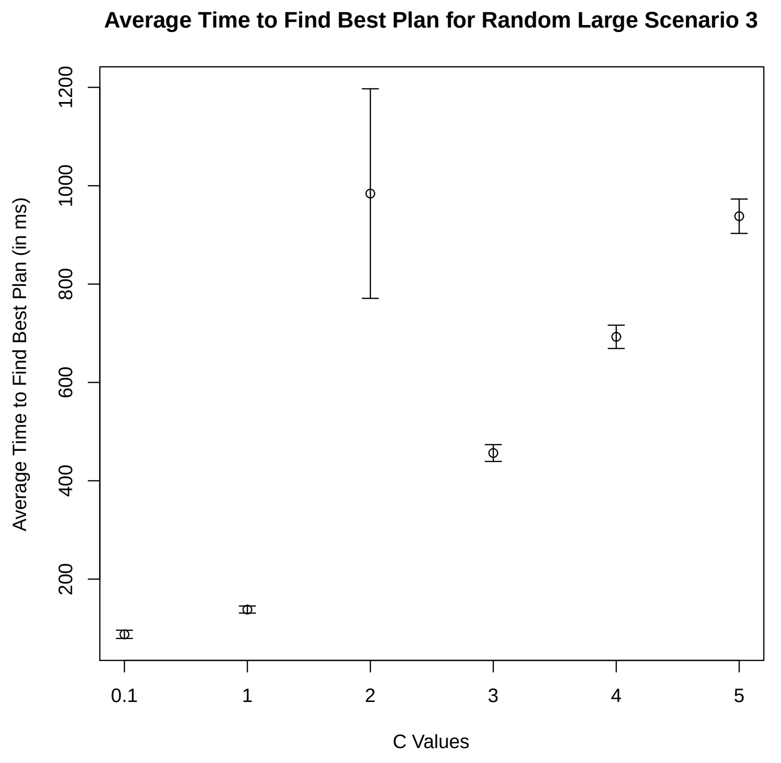


Figure B.7: Average time to find optimal plan (in ms) for Large Random Scenario 3.

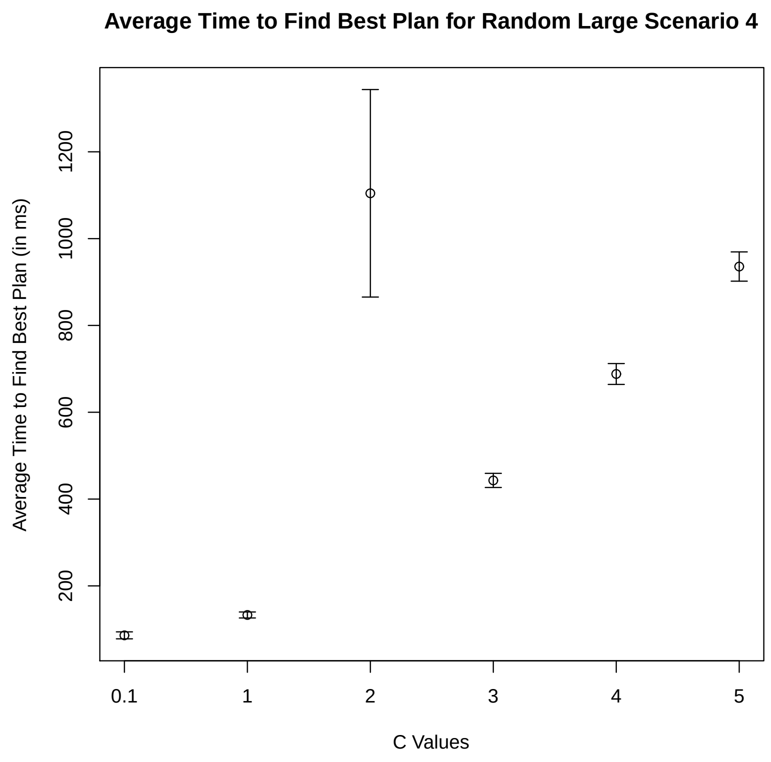
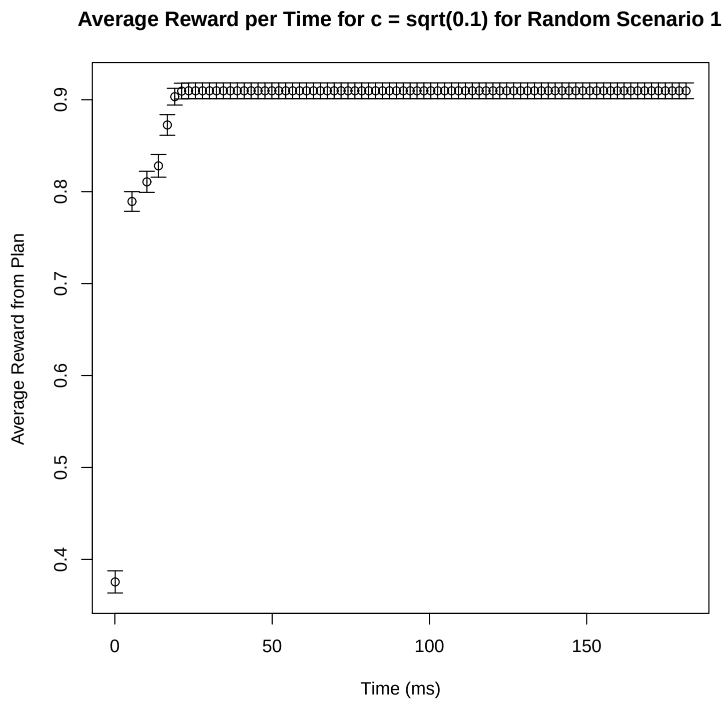


Figure B.8: Average time to find optimal plan (in ms) for Large Random Scenario 4.

Appendix C: Reward Plots

C.1 Random Scenario 1



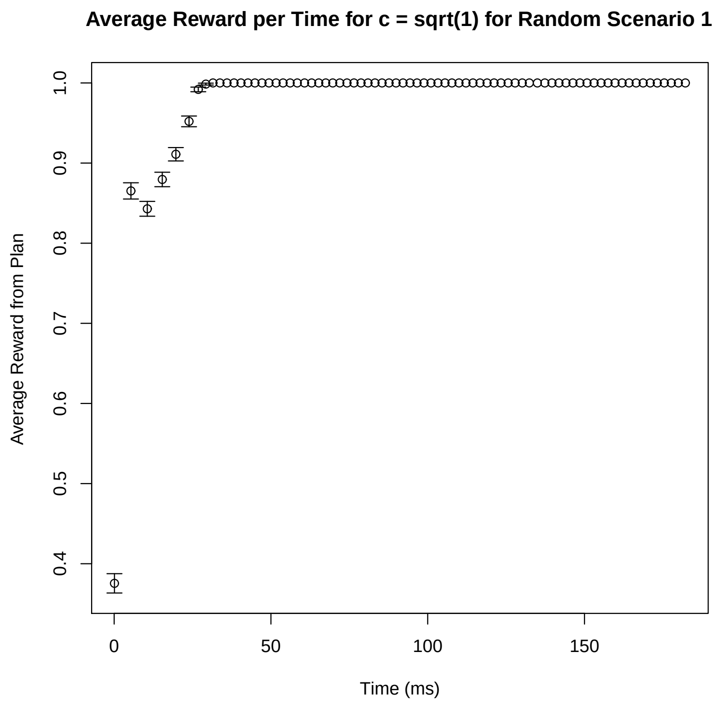


Figure C.1: Average Reward per time for $c = \sqrt{0.1}$ & $c = \sqrt{1}$ for Random Scenario 1.

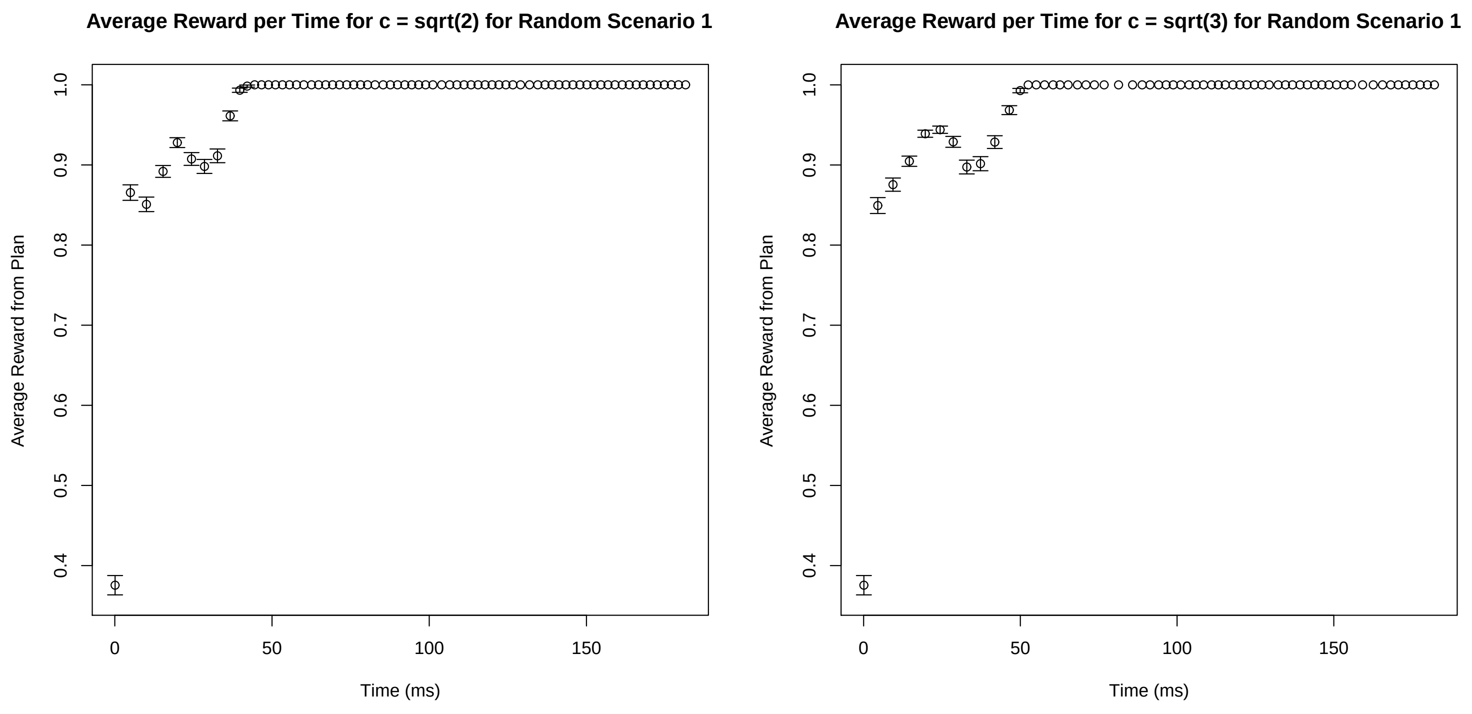


Figure C.2: Average Reward per time for $c = \sqrt{2}$ & $c = \sqrt{3}$ for Random Scenario 1.

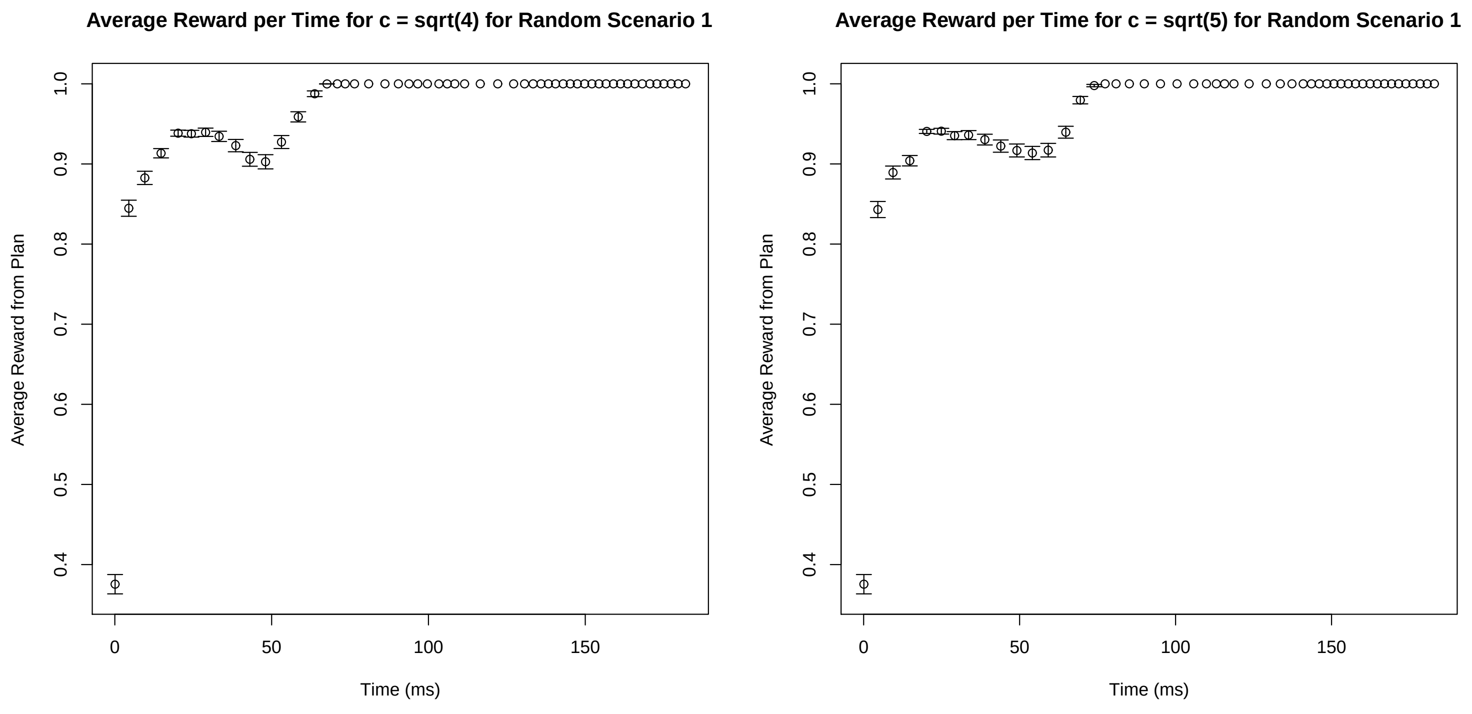
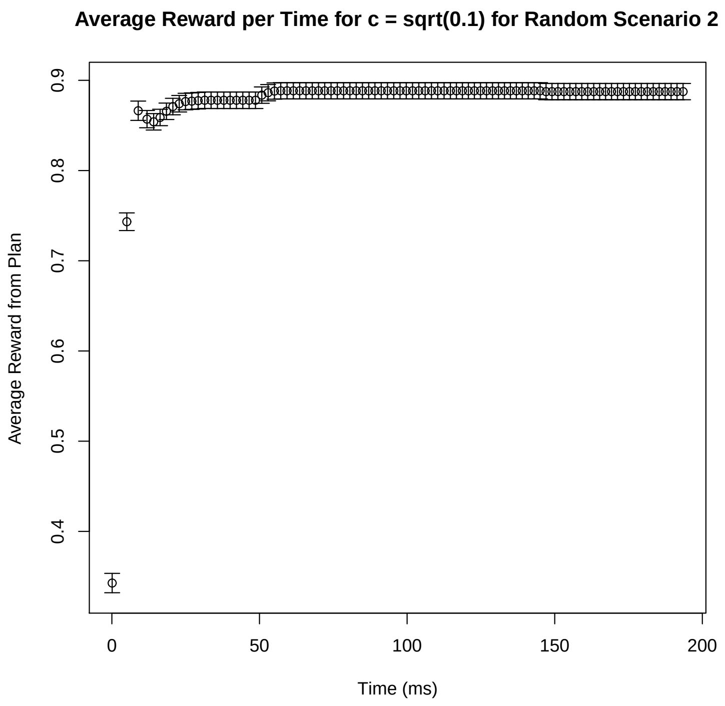


Figure C.3: Average Reward per time for $c = \sqrt{4}$ & $c = \sqrt{5}$ for Random Scenario 1.

C.2 Random Scenario 2



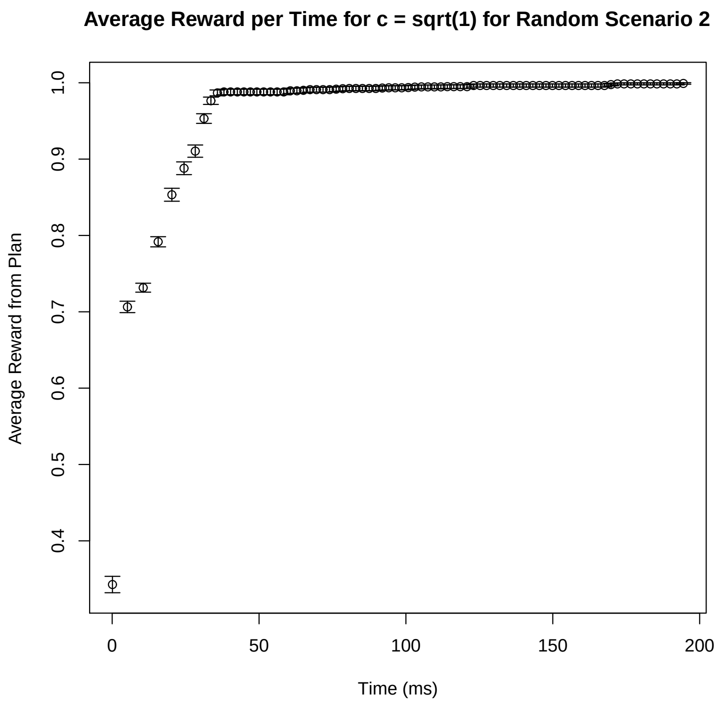
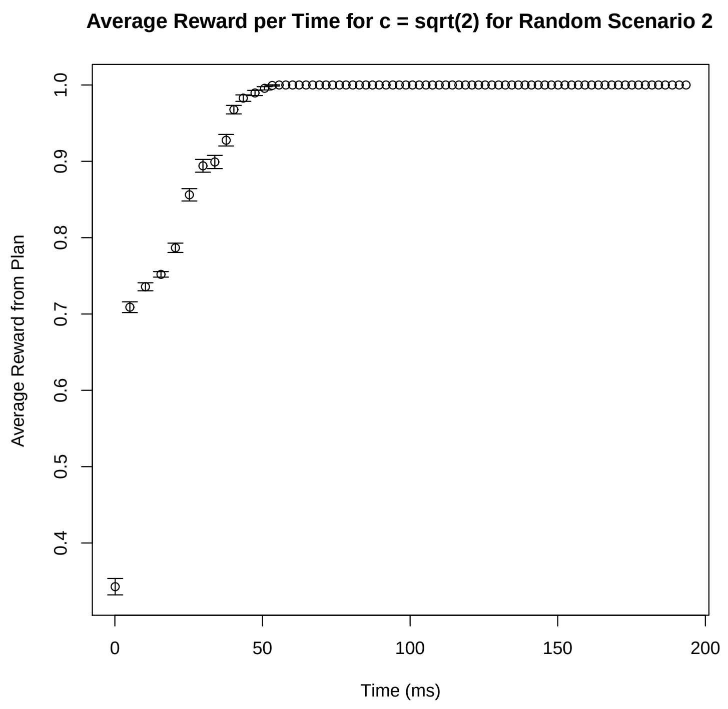


Figure C.4: Average Reward per time for $c = \sqrt{0.1}$ & $c = \sqrt{1}$ for Random Scenario 2.



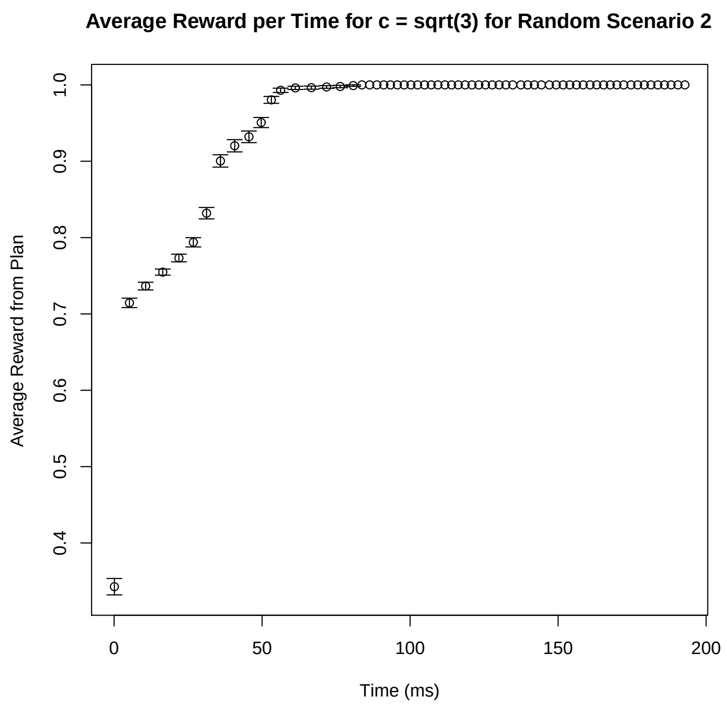
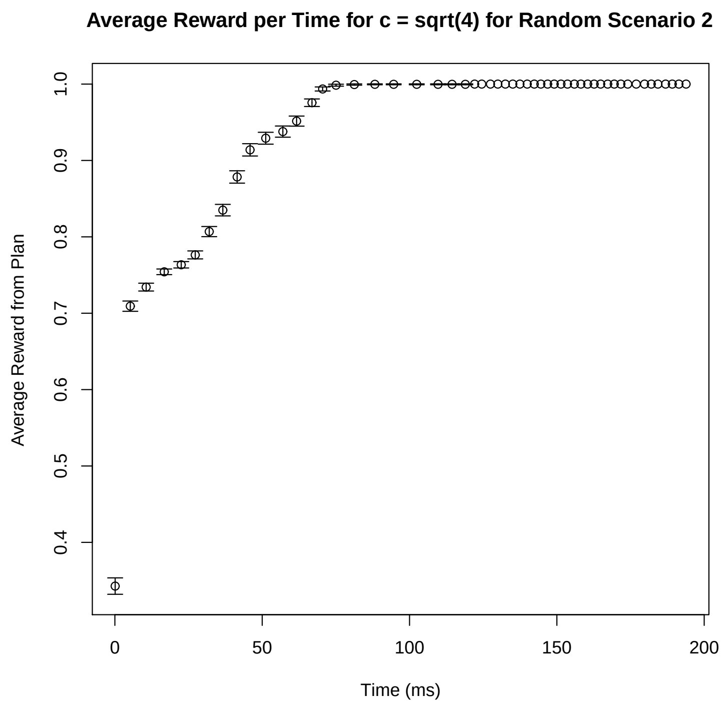


Figure C.5: Average Reward per time for $c = \sqrt{2}$ & $c = \sqrt{3}$ for Random Scenario 2.



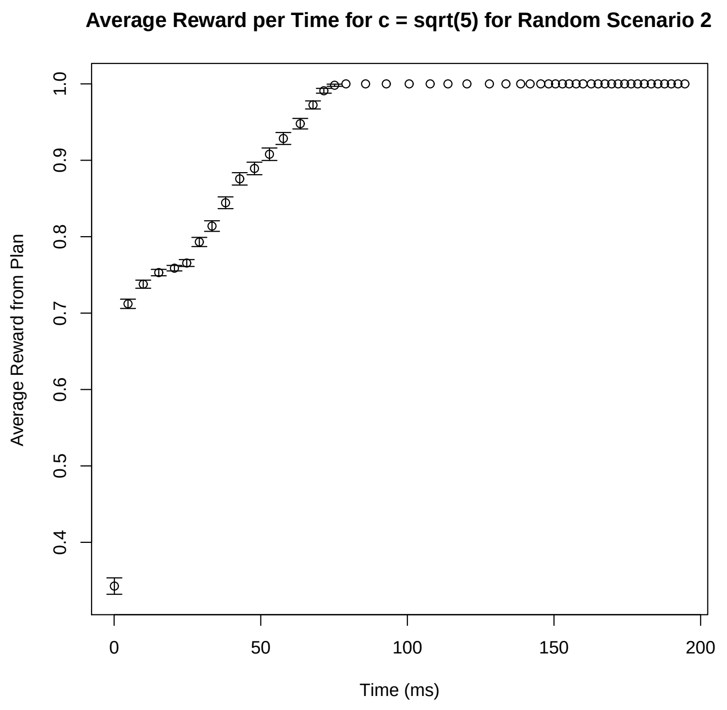


Figure C.6: Average Reward per time for $c = \sqrt{4}$ & $c = \sqrt{5}$ for Random Scenario 2.

C.3 Random Scenario 3

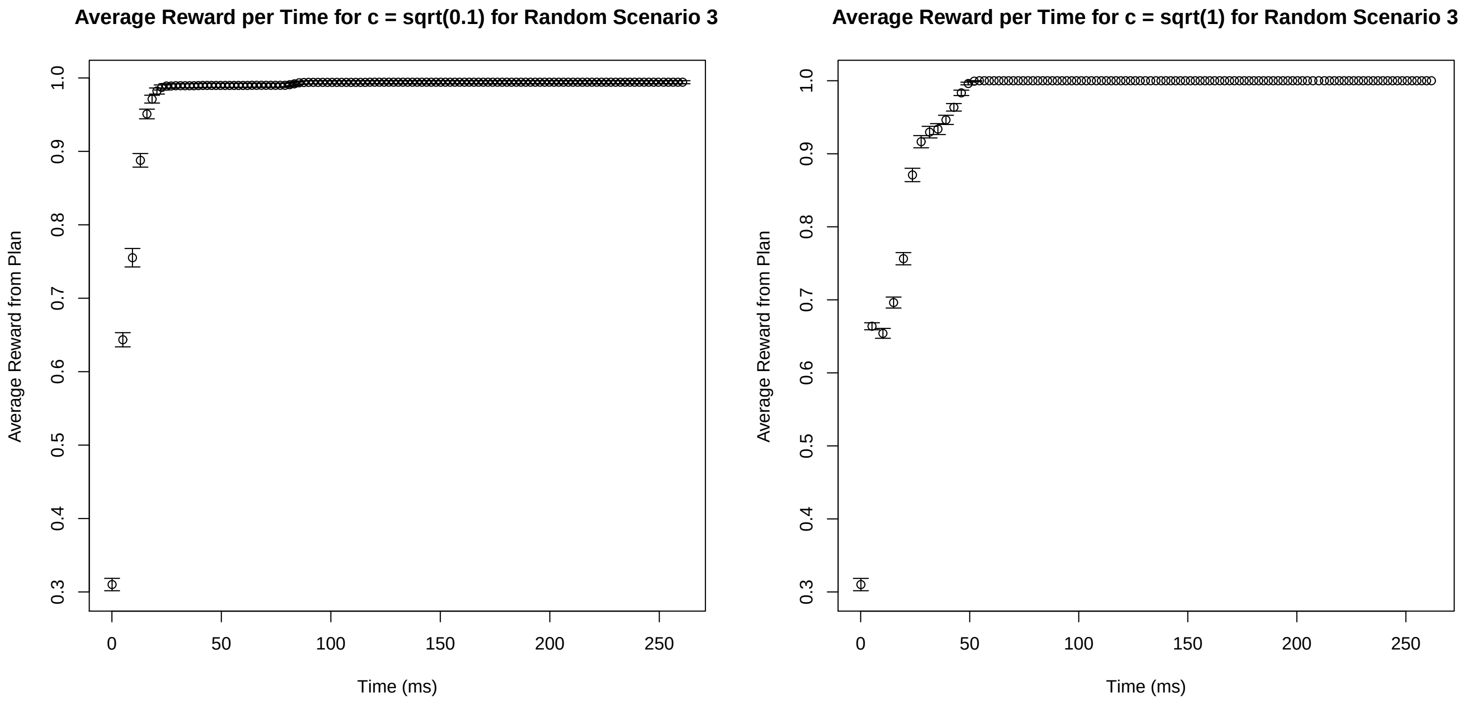


Figure C.7: Average Reward per time for $c = \sqrt{0.1}$ & $c = \sqrt{1}$ for Random Scenario 3.

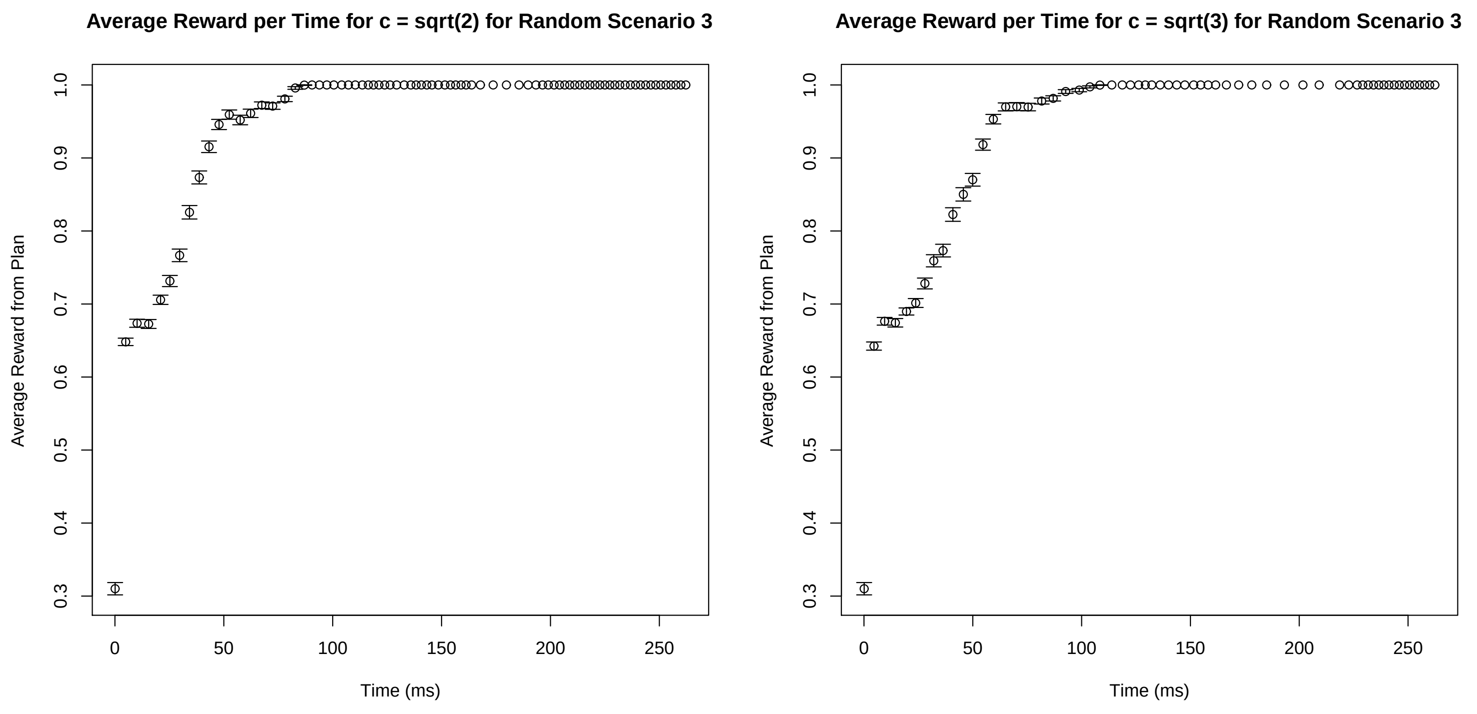


Figure C.8: Average Reward per time for $c = \sqrt{2}$ & $c = \sqrt{3}$ for Random Scenario 3.

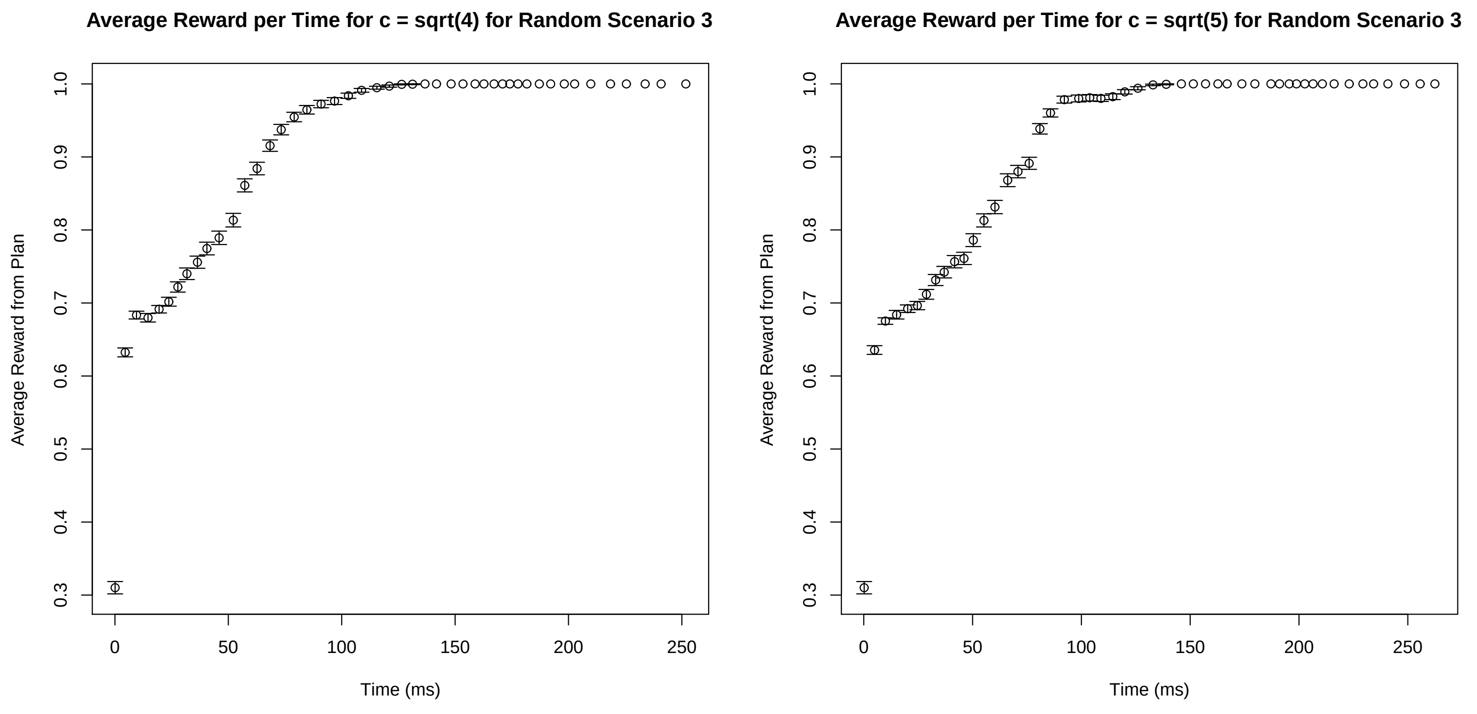
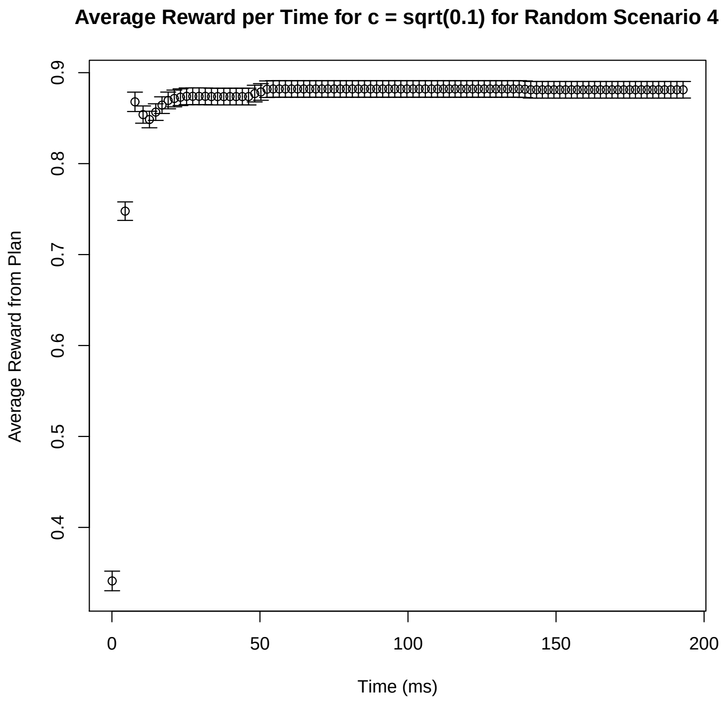


Figure C.9: Average Reward per time for $c = \sqrt{4}$ & $c = \sqrt{5}$ for Random Scenario 3.

C.4 Random Scenario 4



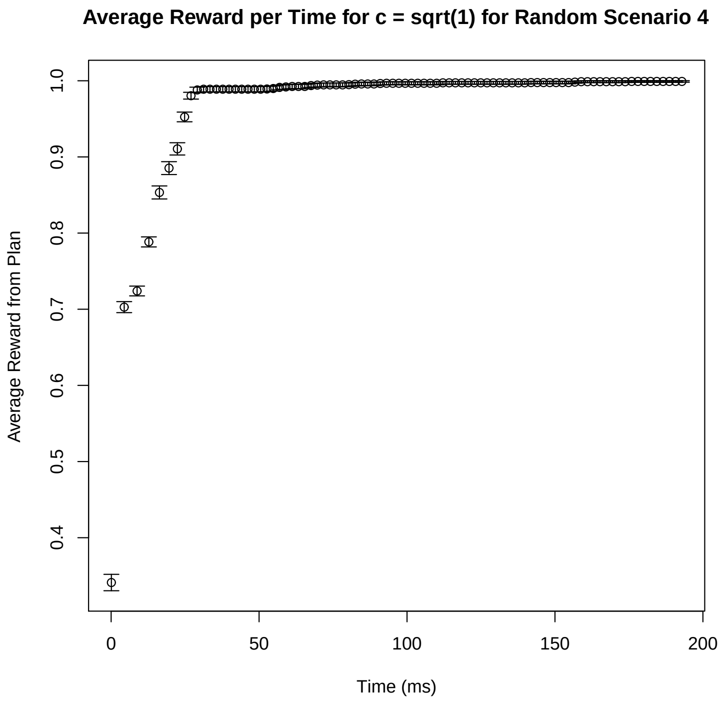
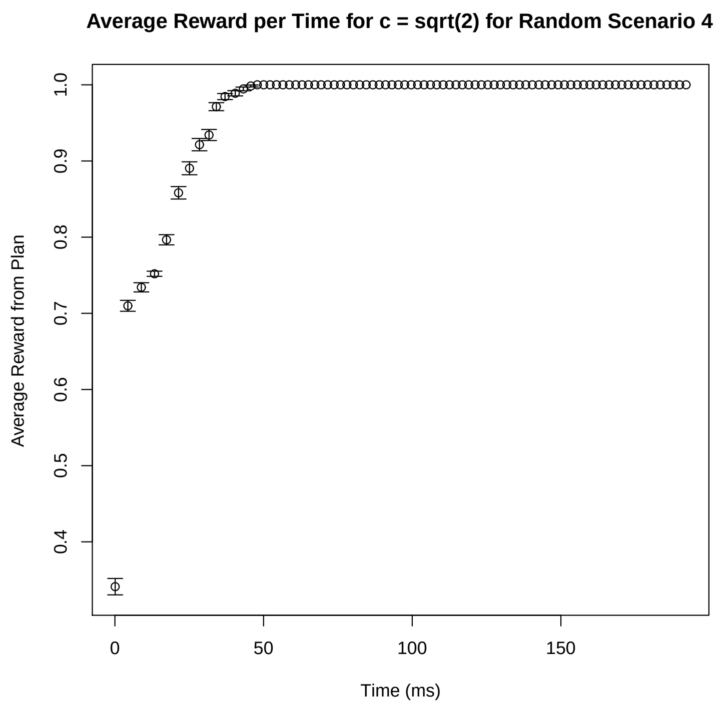


Figure C.10: Average Reward per time for $c = \sqrt{0.1}$ & $c = \sqrt{1}$ for Random Scenario 4.



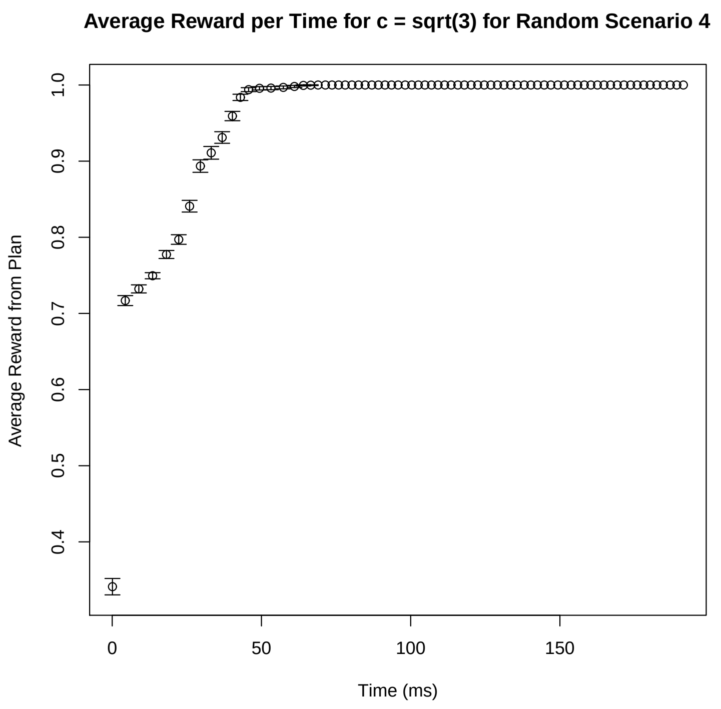
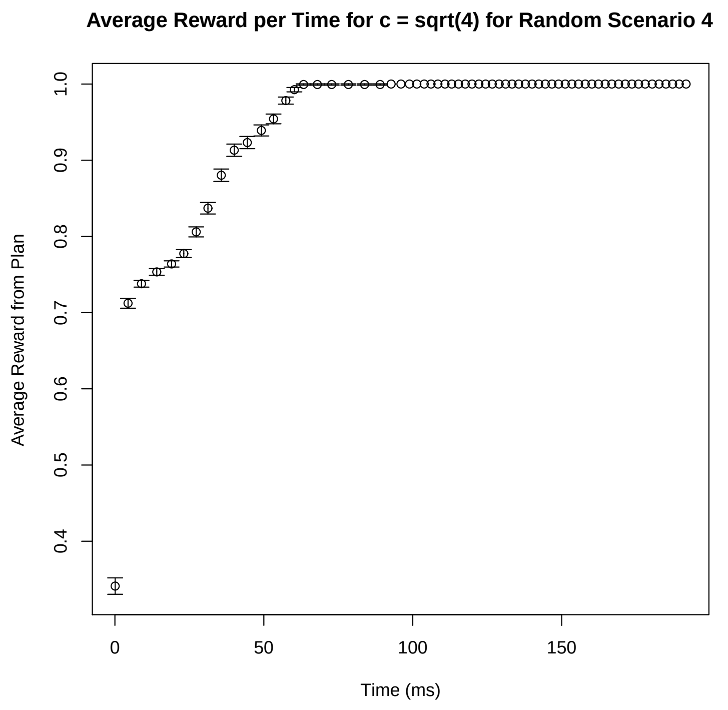


Figure C.11: Average Reward per time for $c = \sqrt{2}$ & $c = \sqrt{3}$ for Random Scenario 4.



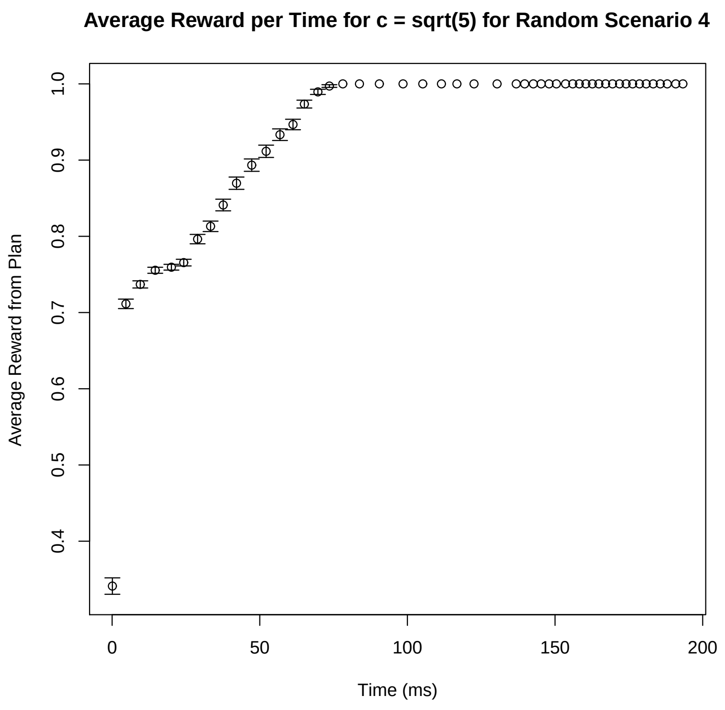
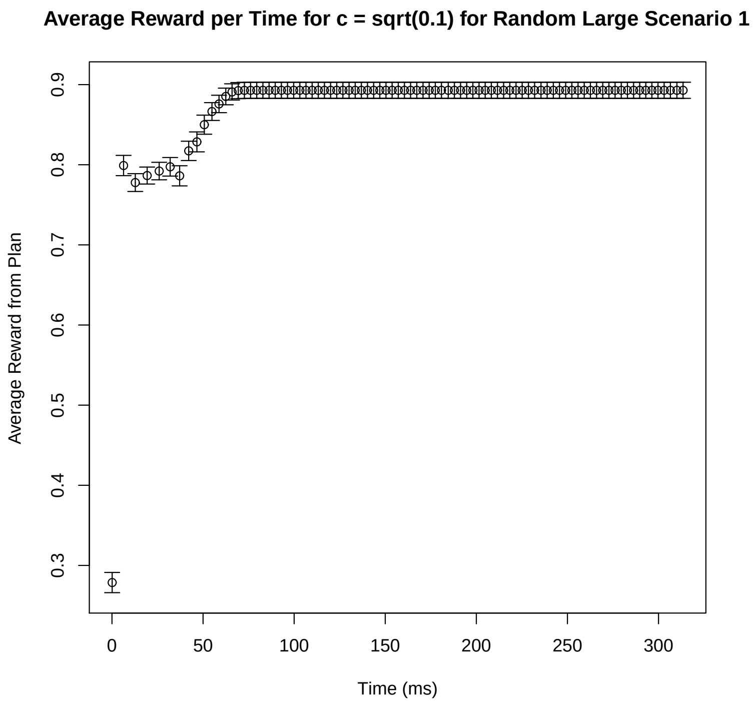


Figure C.12: Average Reward per time for $c = \sqrt{4}$ & $c = \sqrt{5}$ for Random Scenario 4.

C.5 Random Large Scenario 1



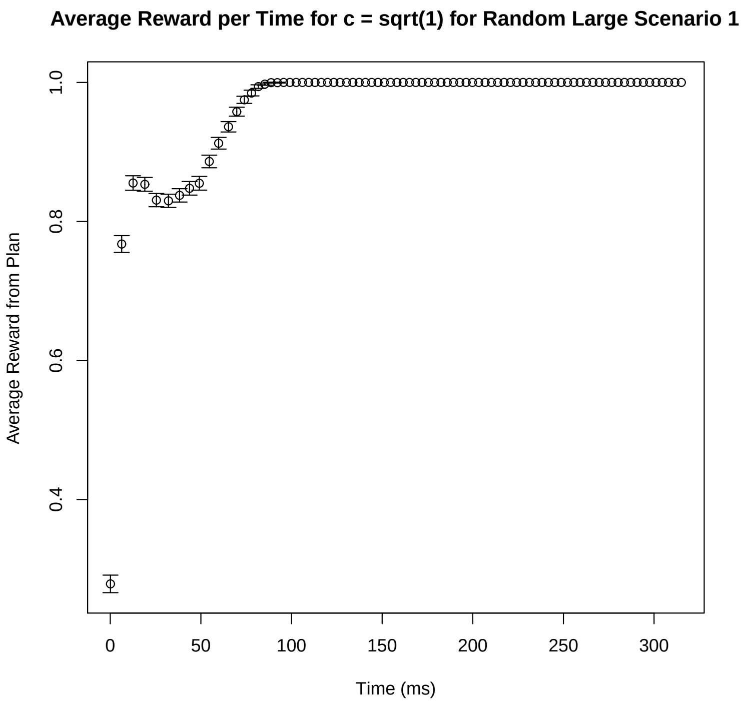


Figure C.13: Average Reward per time for $c = \sqrt{0.1}$ & $c = \sqrt{1}$ for Random Large Scenario 1.

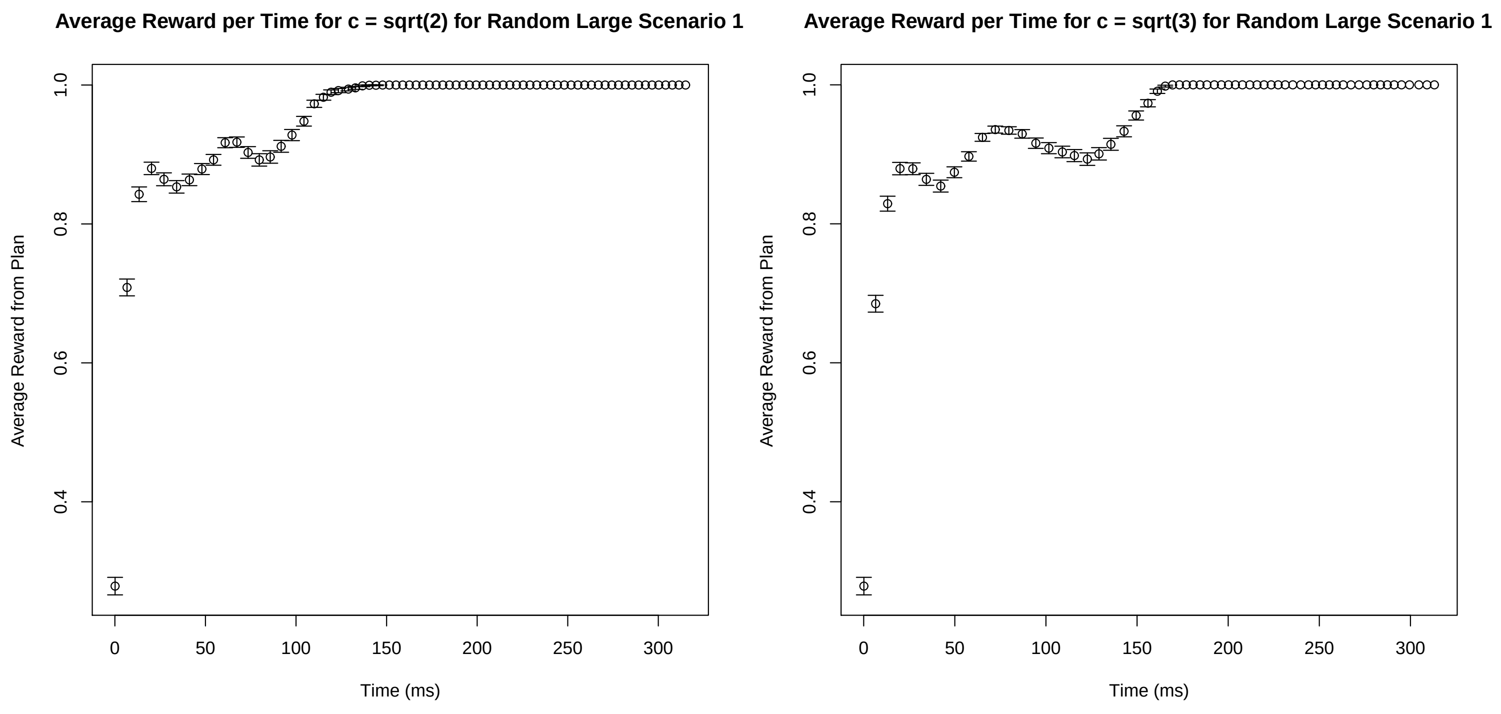


Figure C.14: Average Reward per time for $c = \sqrt{2}$ & $c = \sqrt{3}$ for Random Large Scenario 1.

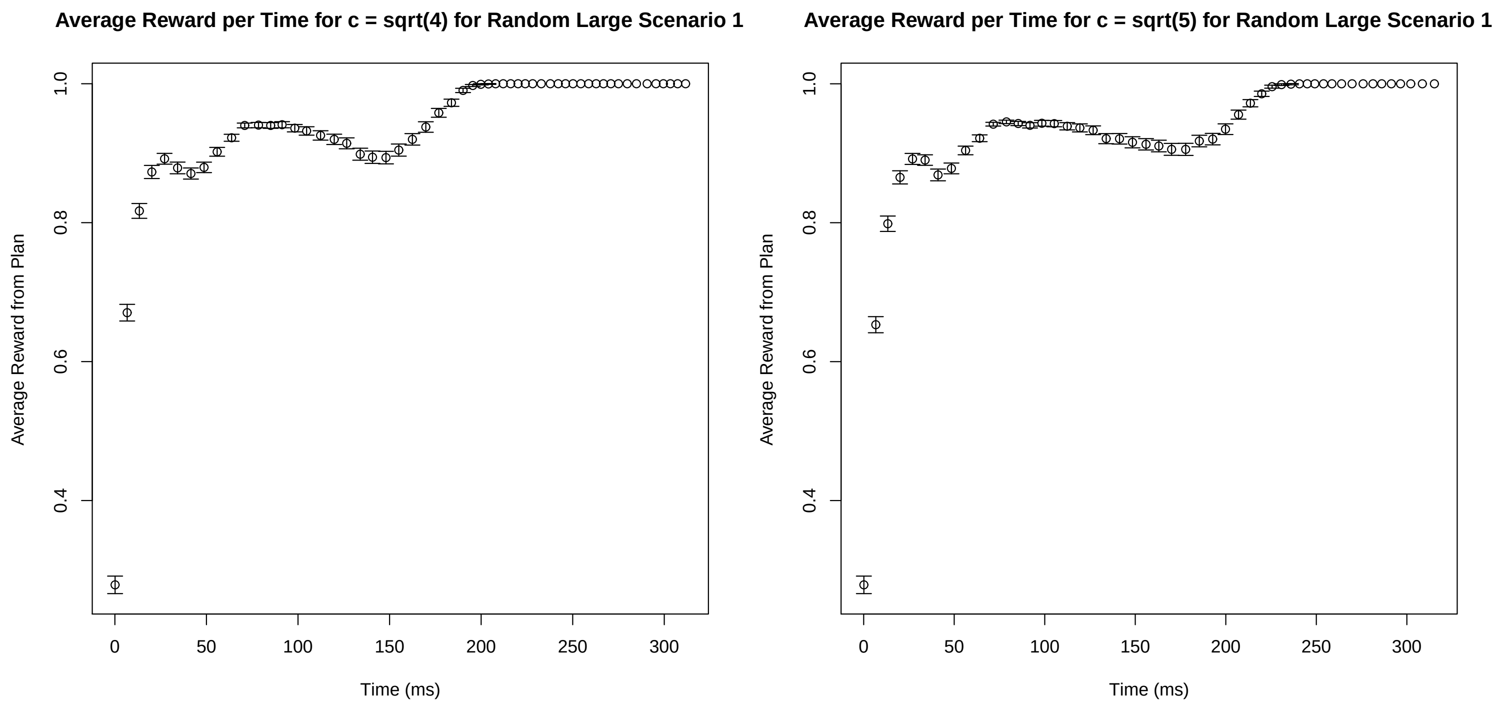
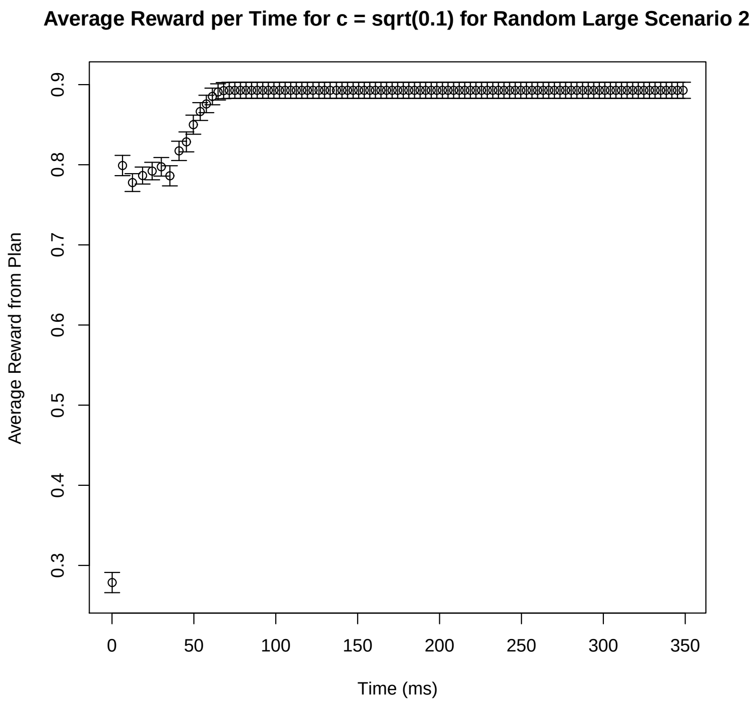


Figure C.15: Average Reward per time for $c = \sqrt{4}$ & $c = \sqrt{5}$ for Random Large Scenario 1.

C.6 Random Large Scenario 2



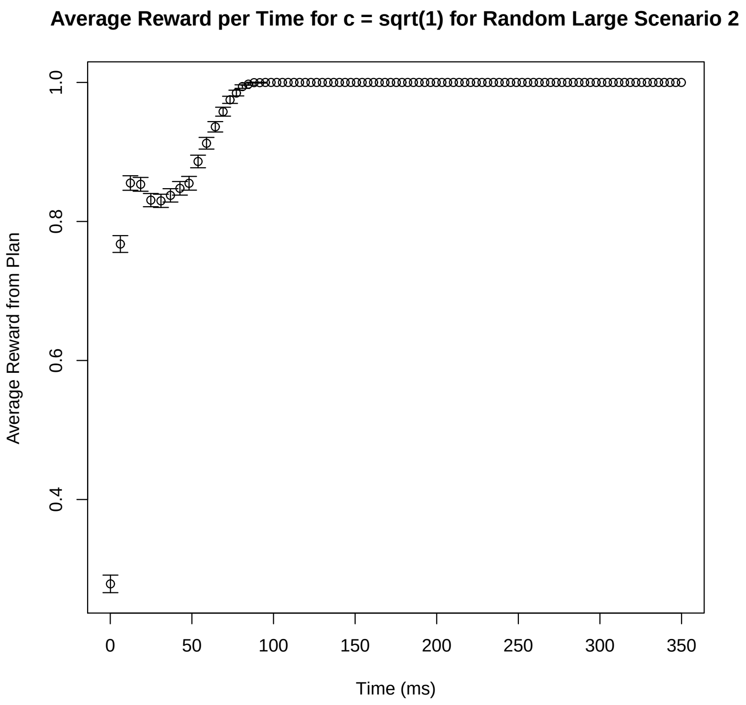
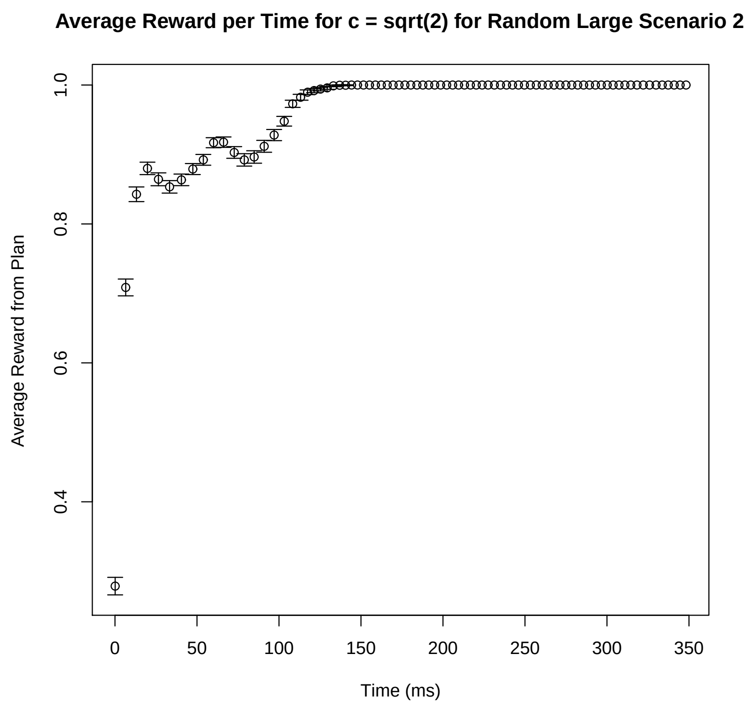


Figure C.16: Average Reward per time for $c = \sqrt{0.1}$ & $c = \sqrt{1}$ for Random Large Scenario 2.



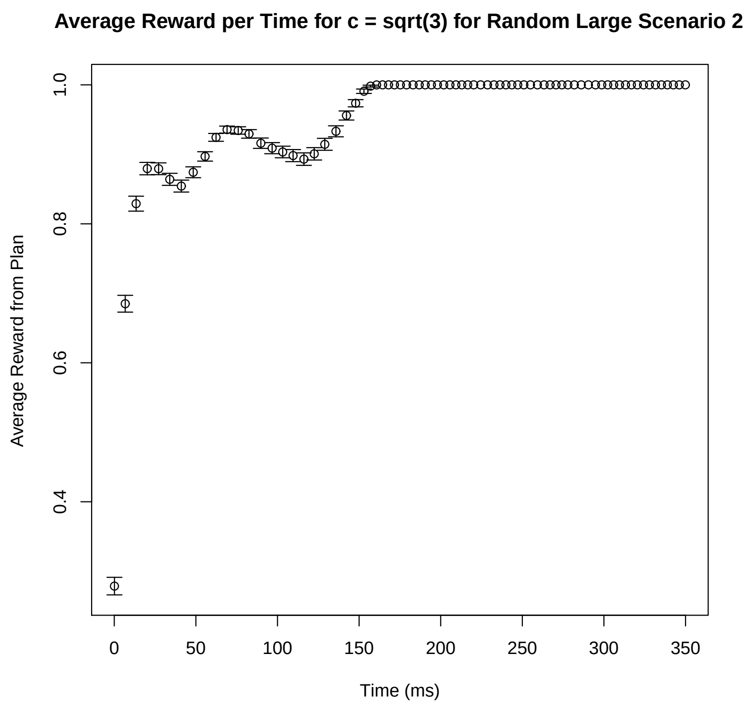
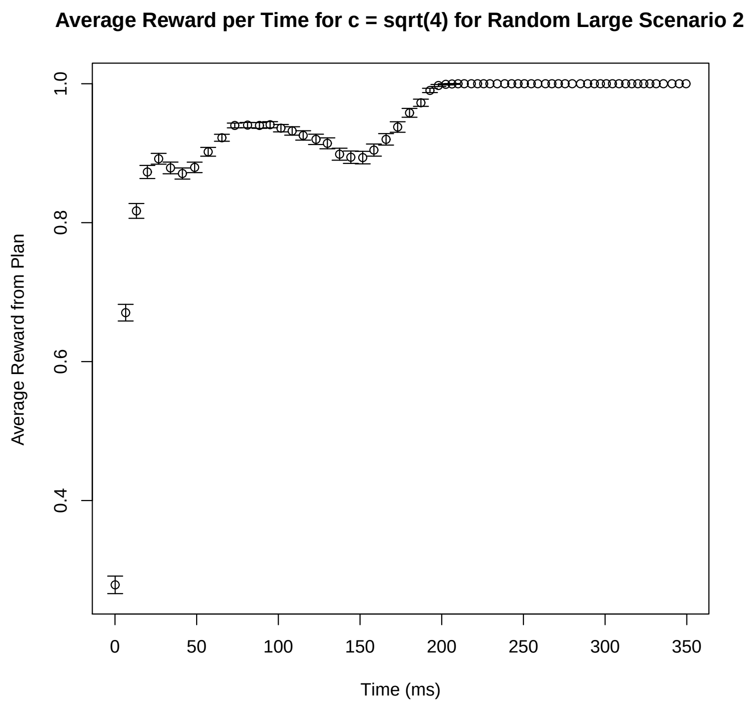


Figure C.17: Average Reward per time for $c = \sqrt{2}$ & $c = \sqrt{3}$ for Random Large Scenario 2.



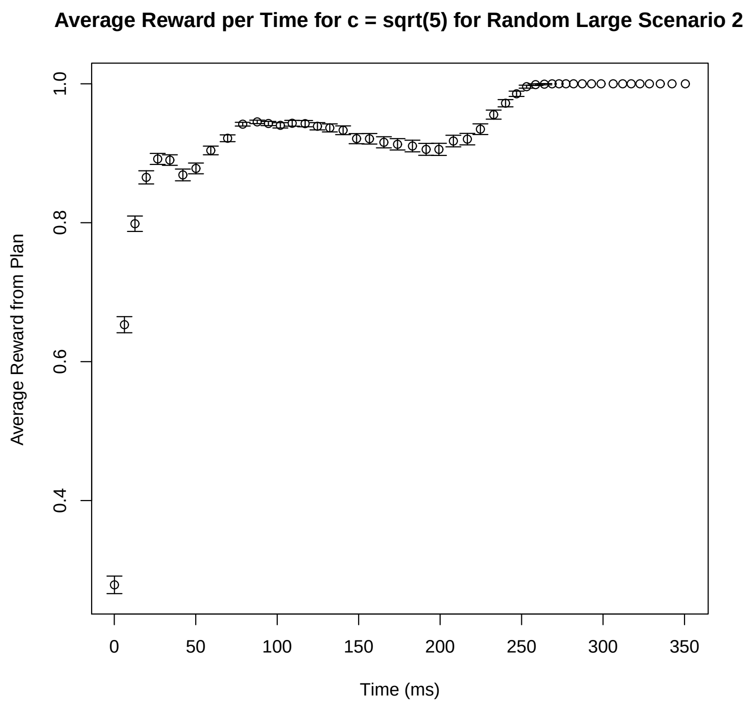
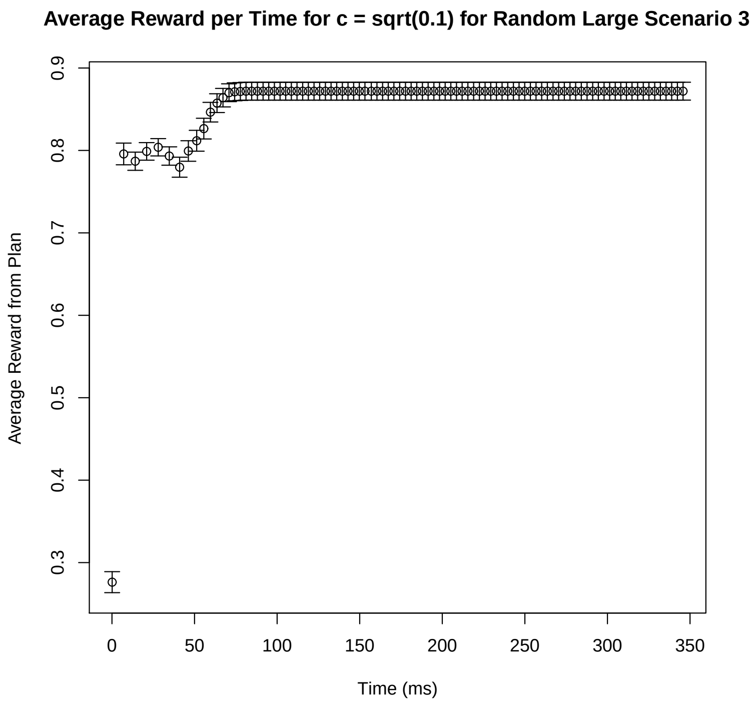


Figure C.18: Average Reward per time for $c = \sqrt{4}$ & $c = \sqrt{5}$ for Random Large Scenario 2.

C.7 Random Large Scenario 3



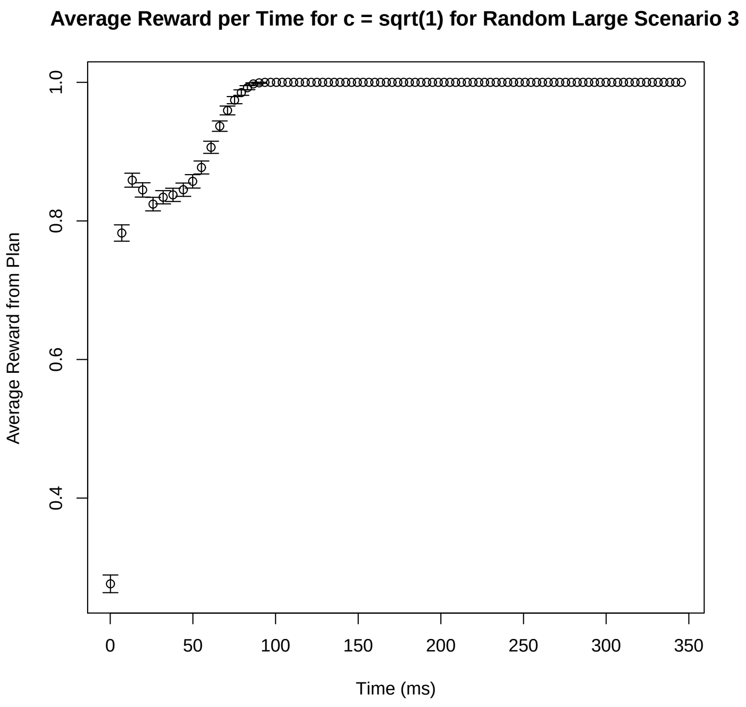


Figure C.19: Average Reward per time for $c = \sqrt{0.1}$ & $c = \sqrt{1}$ for Random Large Scenario 3.

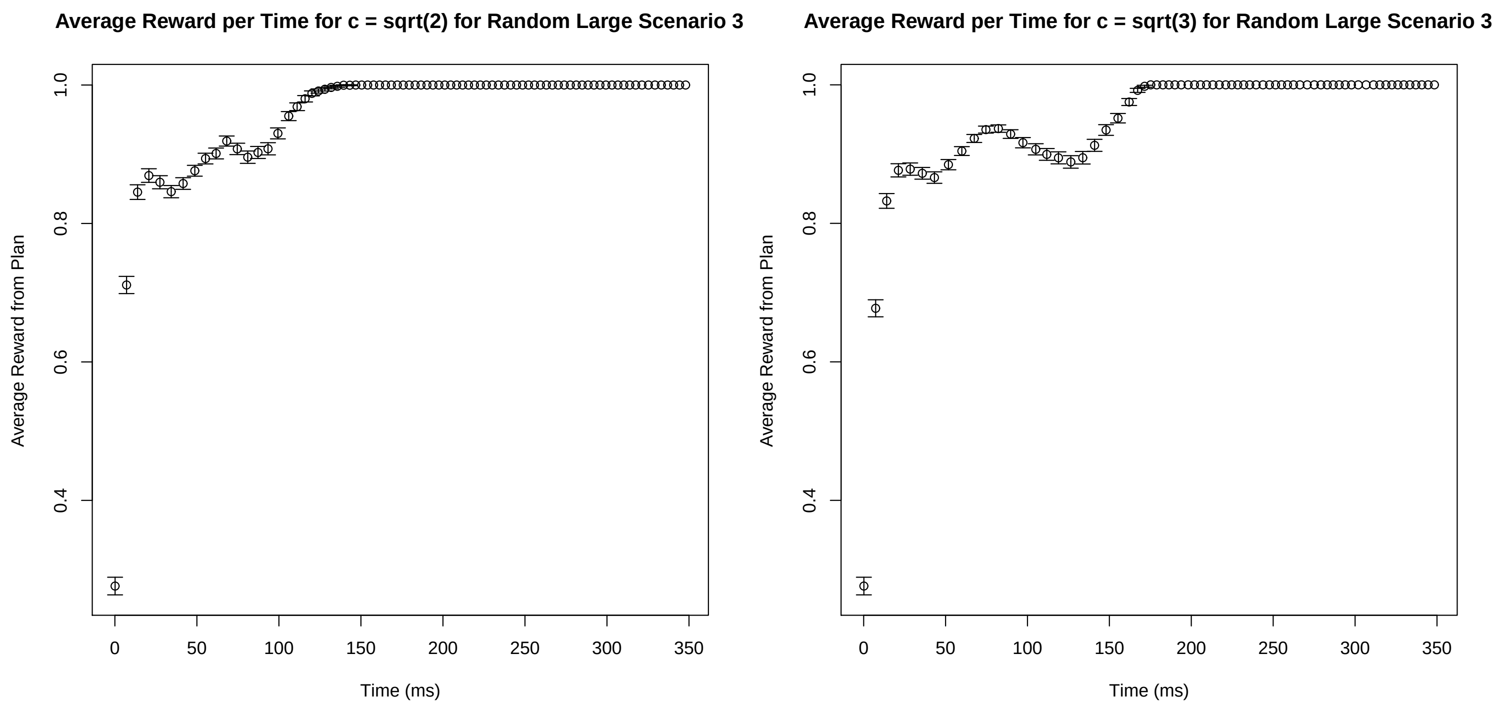


Figure C.20: Average Reward per time for $c = \sqrt{2}$ & $c = \sqrt{3}$ for Random Large Scenario 3.

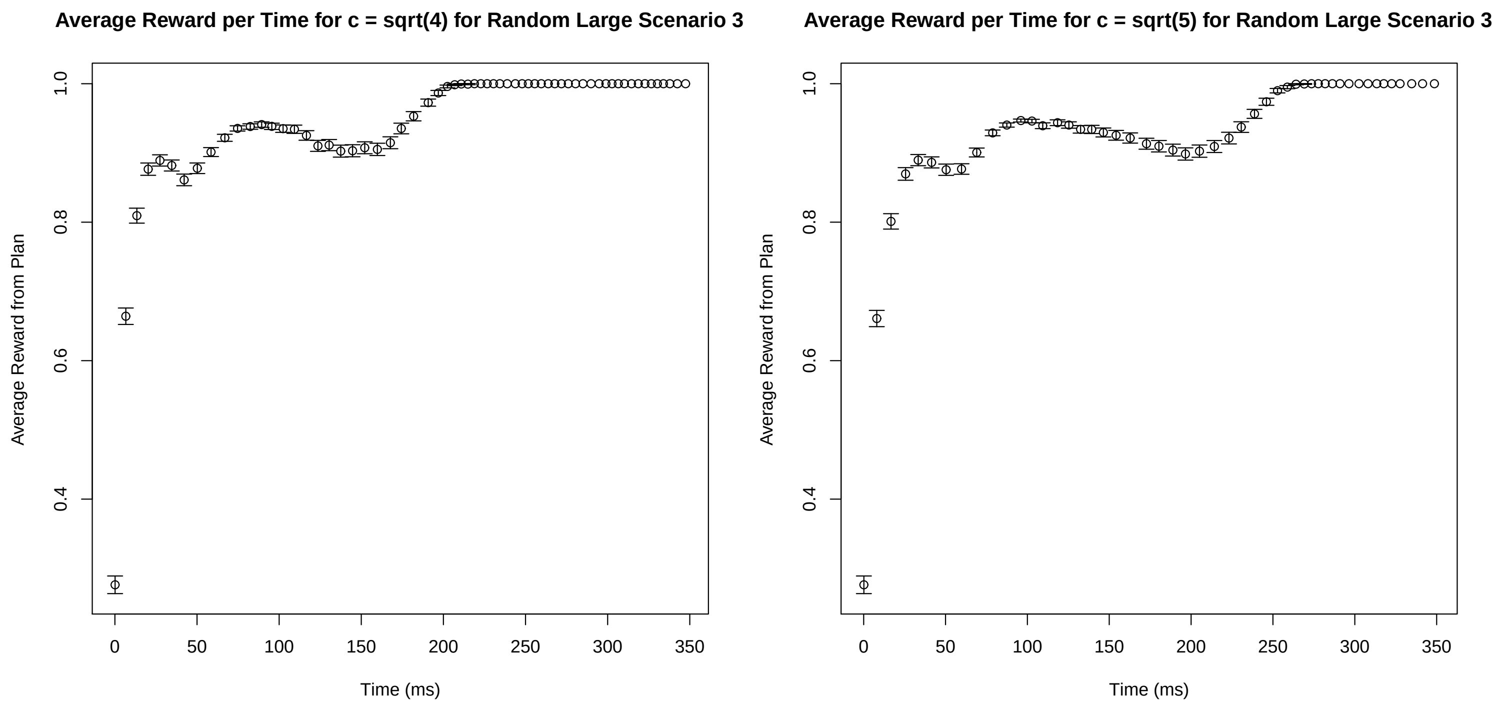
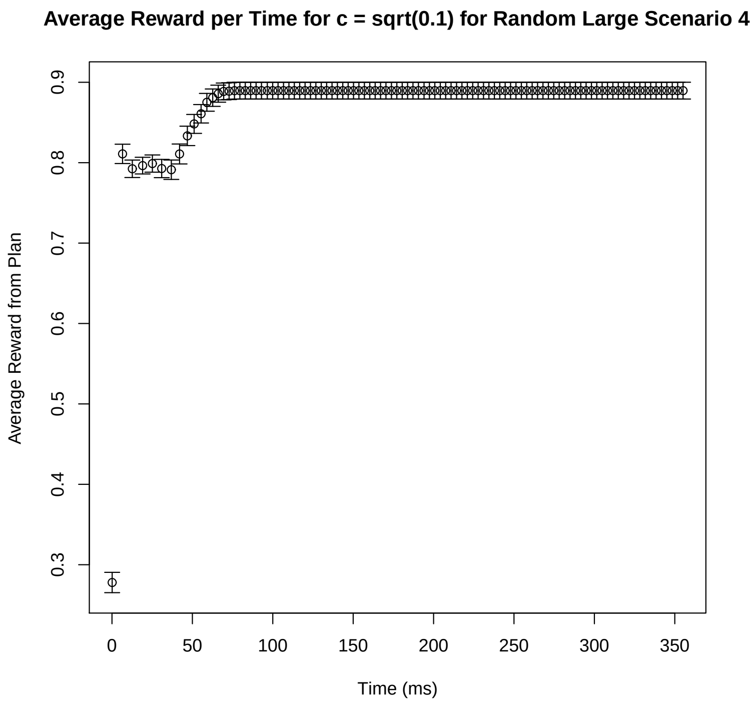


Figure C.21: Average Reward per time for $c = \sqrt{4}$ & $c = \sqrt{5}$ for Random Large Scenario 3.

C.8 Random Large Scenario 4



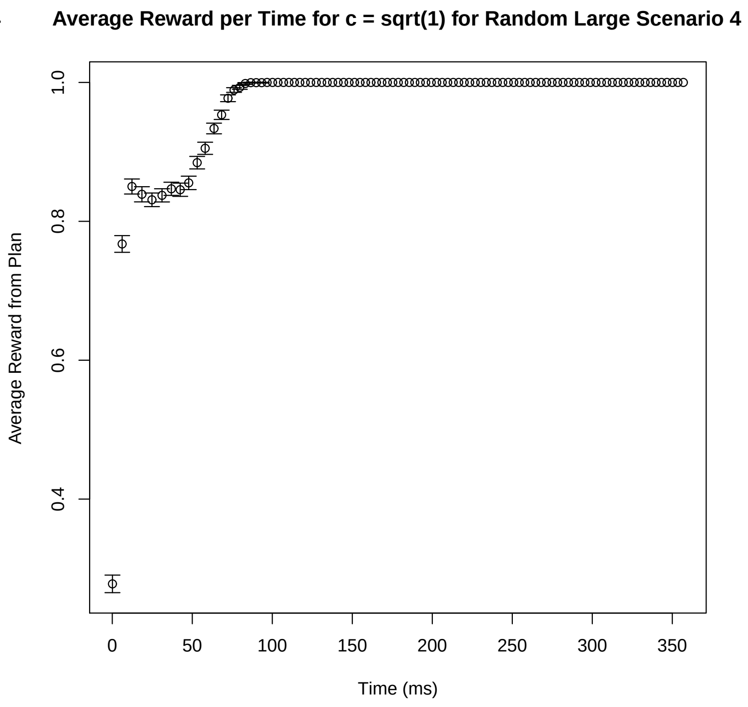


Figure C.22: Average Reward per time for $c = \sqrt{0.1}$ & $c = \sqrt{1}$ for Random Large Scenario 4.

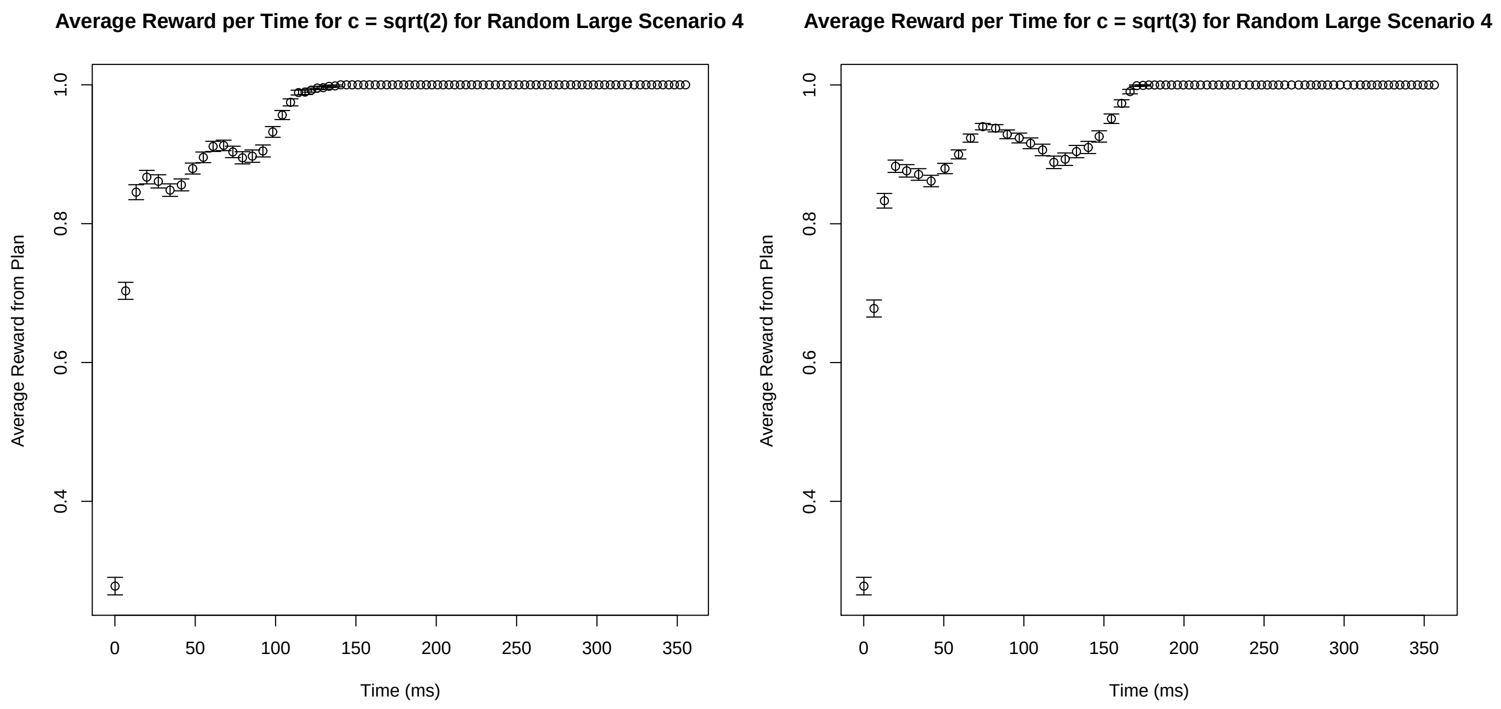


Figure C.23: Average Reward per time for $c = \sqrt{2}$ & $c = \sqrt{3}$ for Random Large Scenario 4.

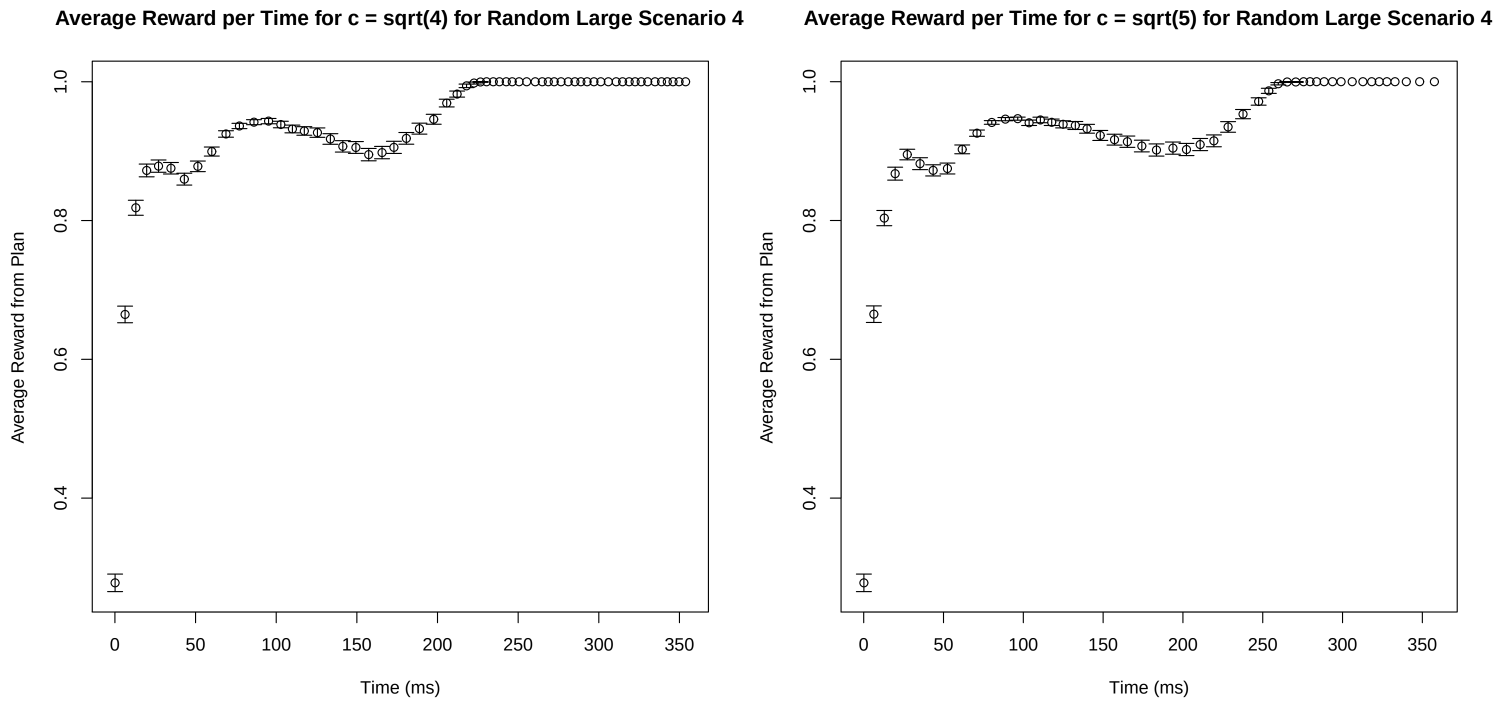
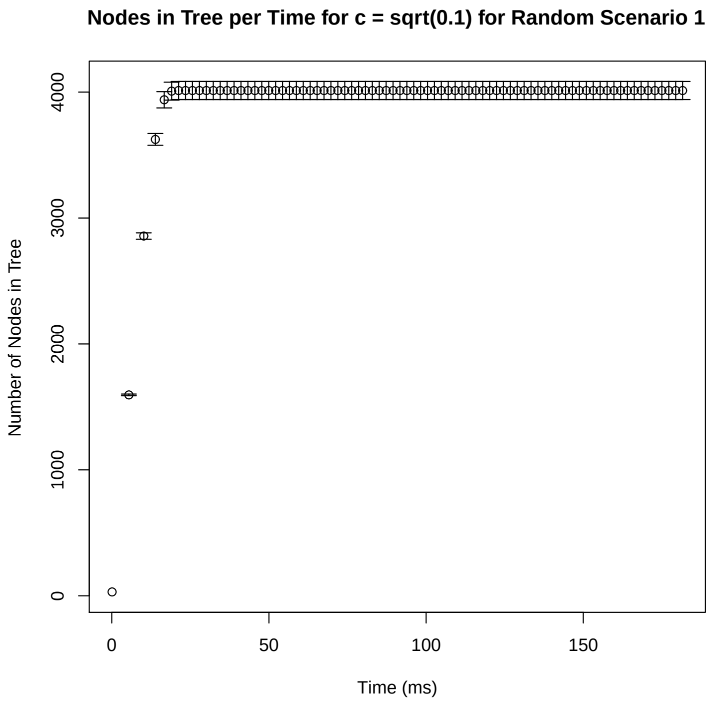


Figure C.24: Average Reward per time for $c = \sqrt{4}$ & $c = \sqrt{5}$ for Random Large Scenario 4.

Appendix D: Nodes Plots

D.1 Random Scenario 1



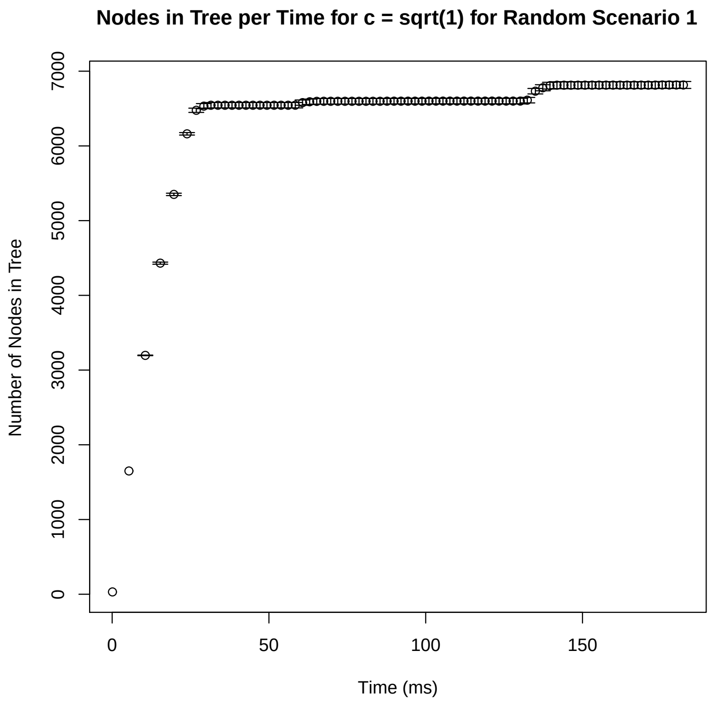
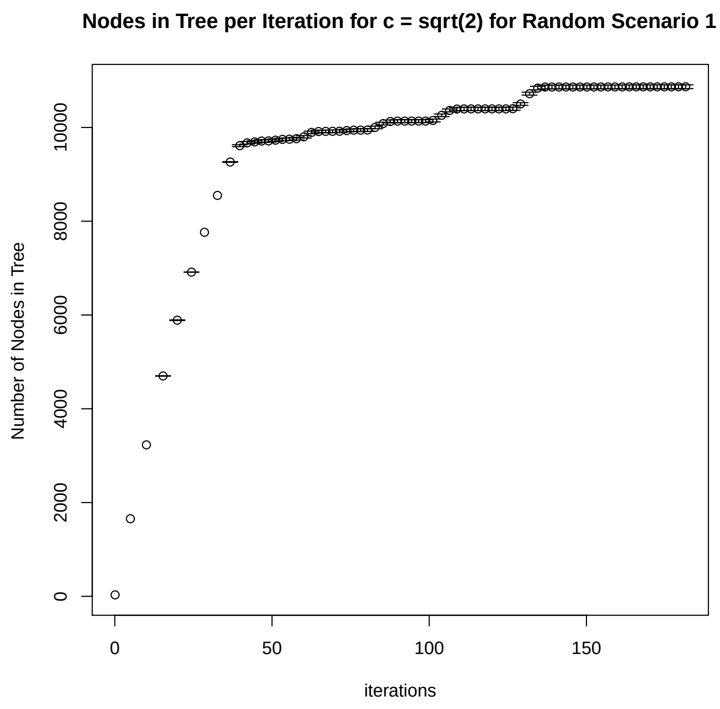


Figure D.1: Average Nodes per time for $c = \sqrt{0.1}$ & $c = \sqrt{1}$ for Random Scenario 1.



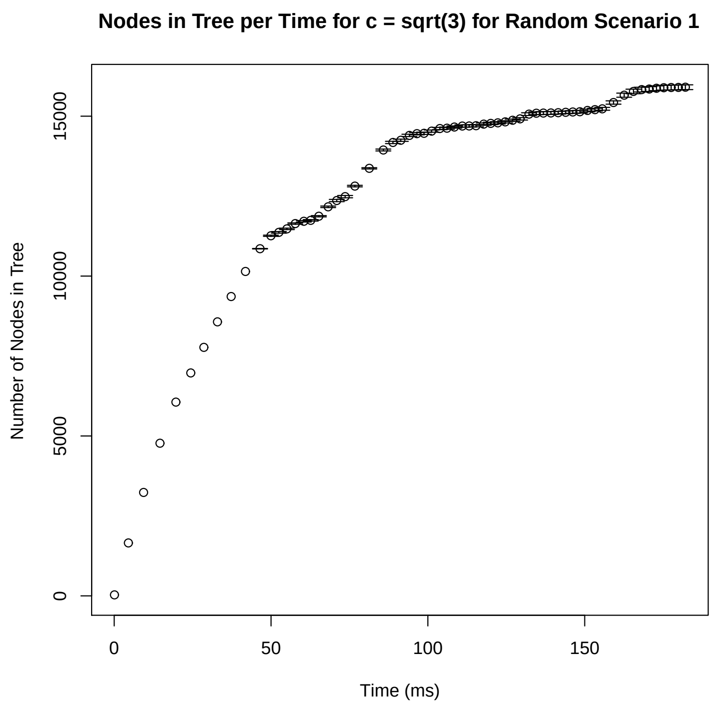
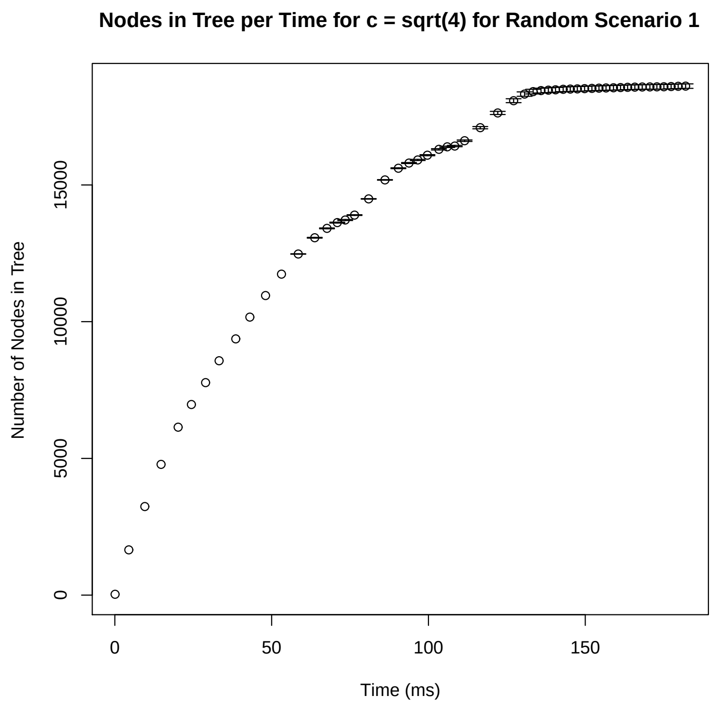


Figure D.2: Average Nodes per time for $c = \sqrt{2}$ & $c = \sqrt{3}$ for Random Scenario 1.



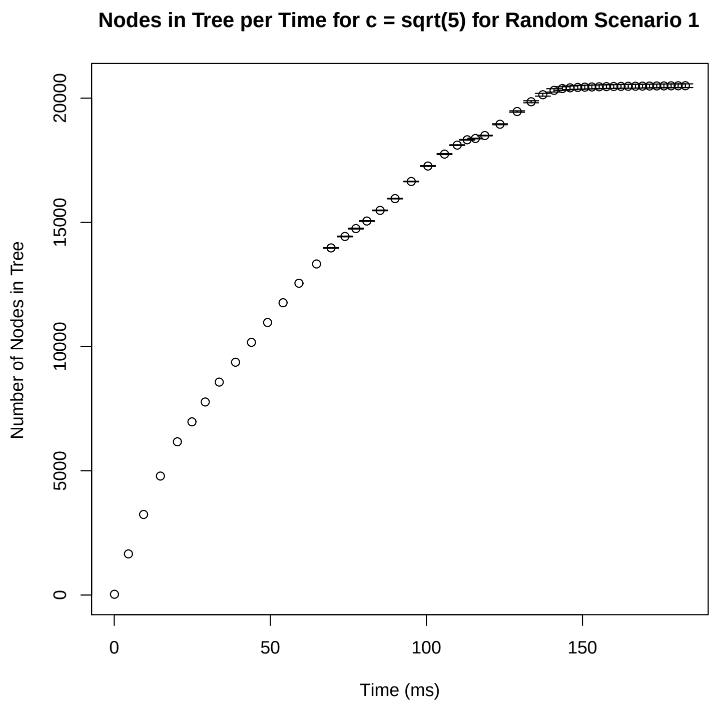
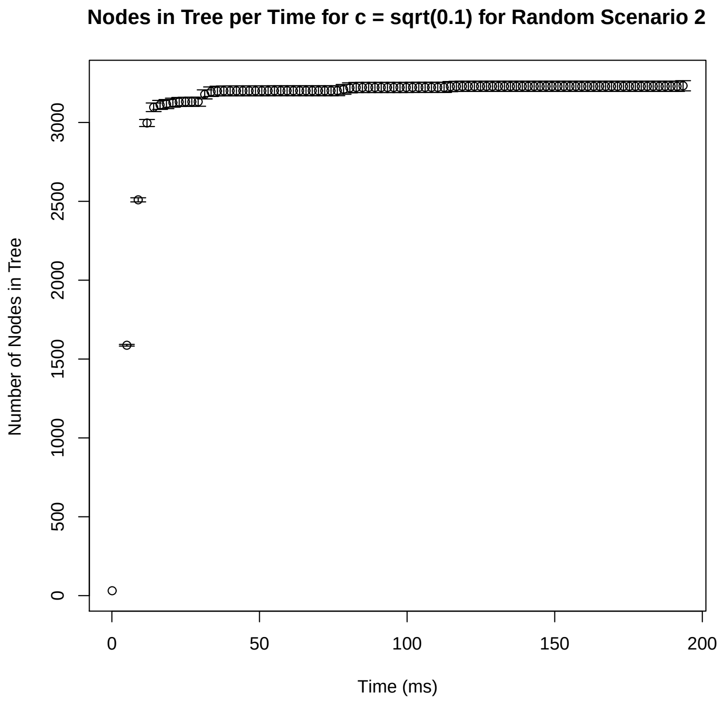


Figure D.3: Average Nodes per time for $c = \sqrt{4}$ & $c = \sqrt{5}$ for Random Scenario 1.

D.2 Random Scenario 2



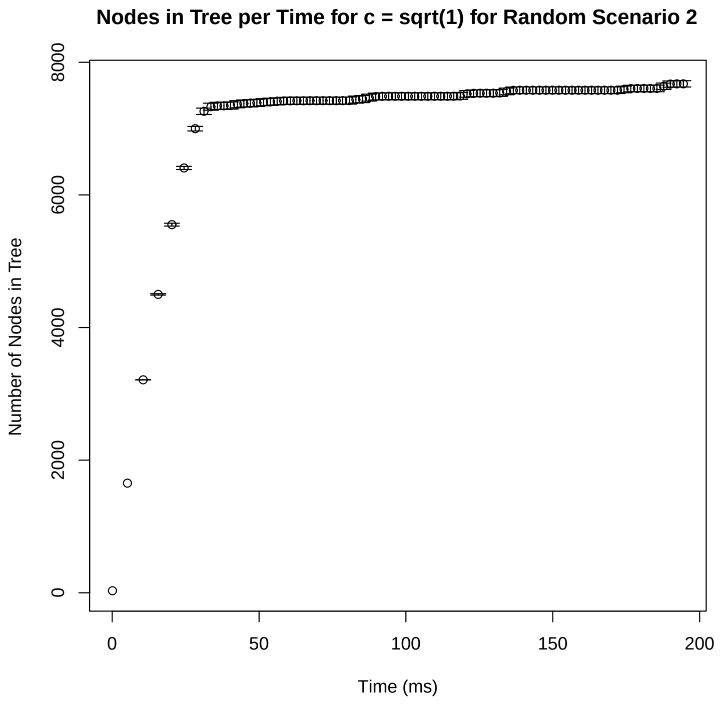
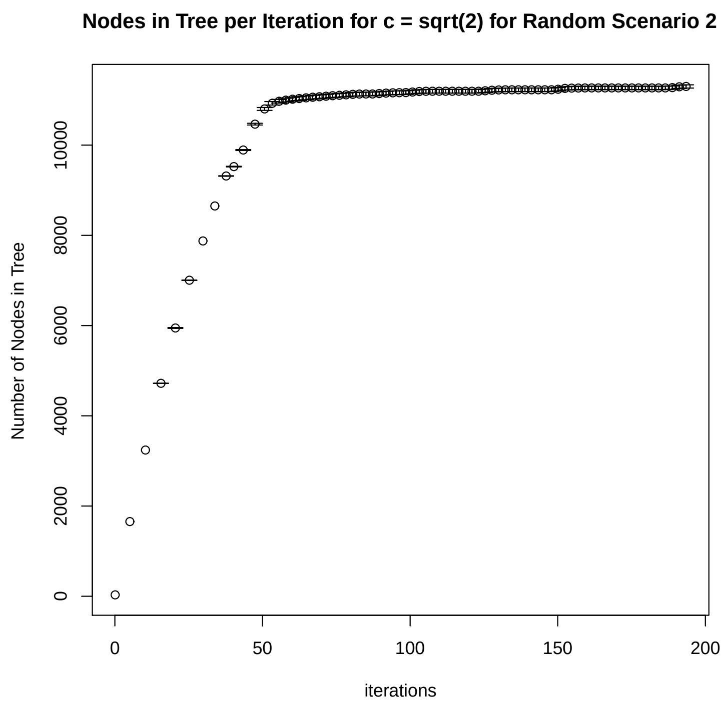


Figure D.4: Average Nodes per time for $c = \sqrt{0.1}$ & $c = \sqrt{1}$ for Random Scenario 2.



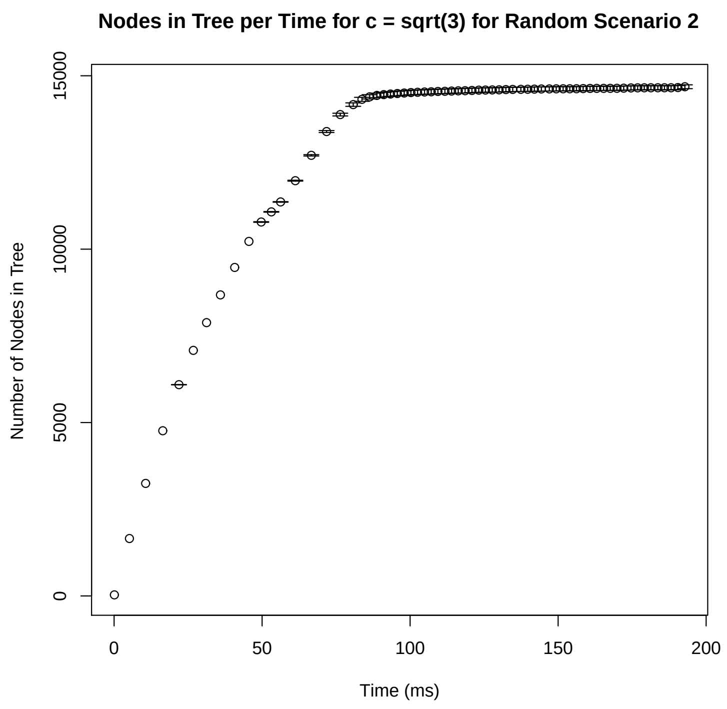
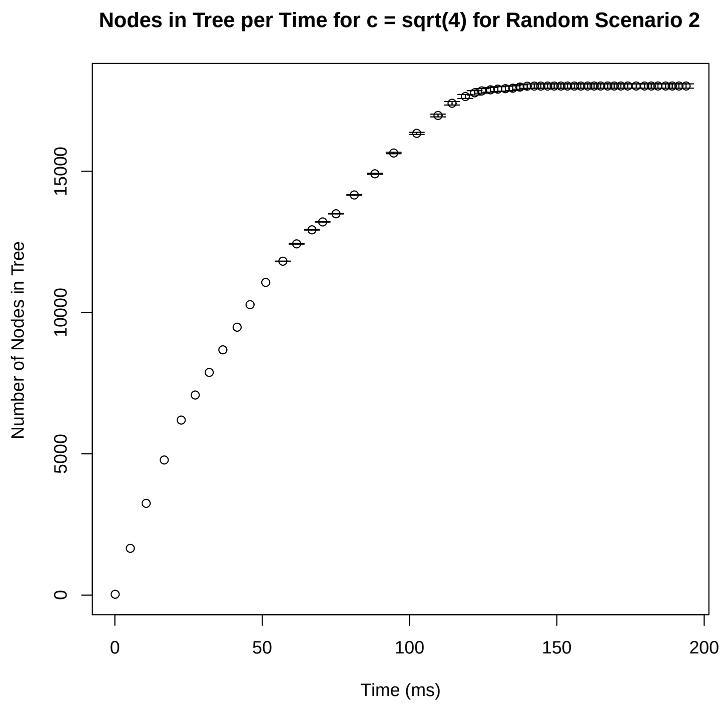


Figure D.5: Average Nodes per time for $c = \sqrt{2}$ & $c = \sqrt{3}$ for Random Scenario 2.



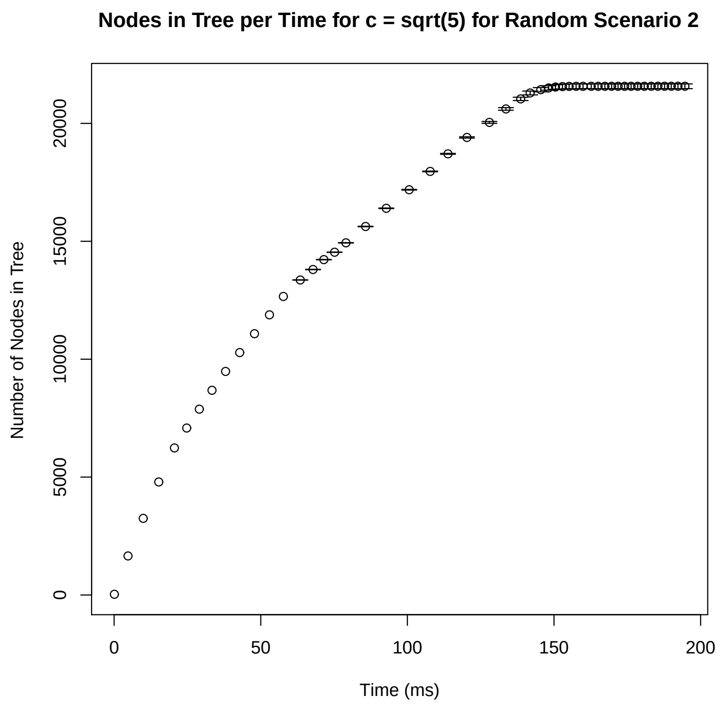
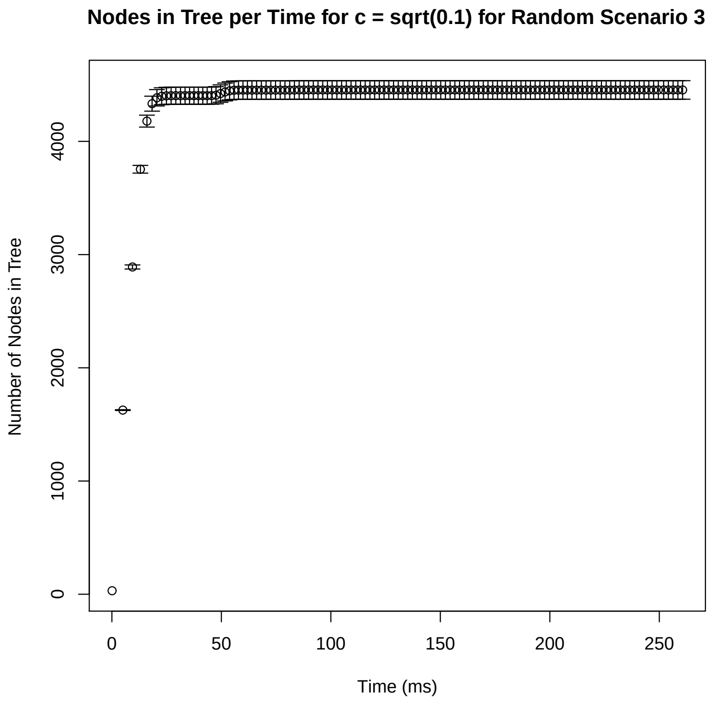


Figure D.6: Average Nodes per time for $c = \sqrt{4}$ & $c = \sqrt{5}$ for Random Scenario 2.

D.3 Random Scenario 3



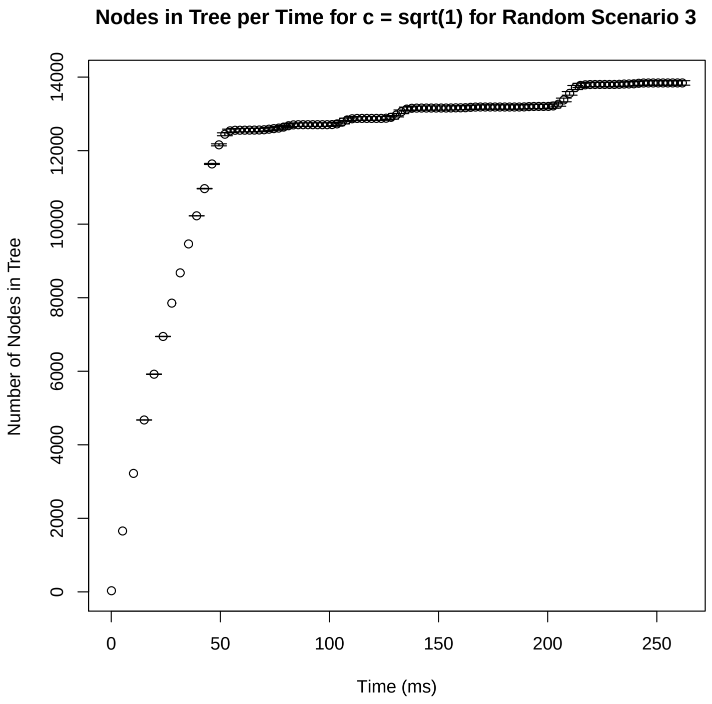
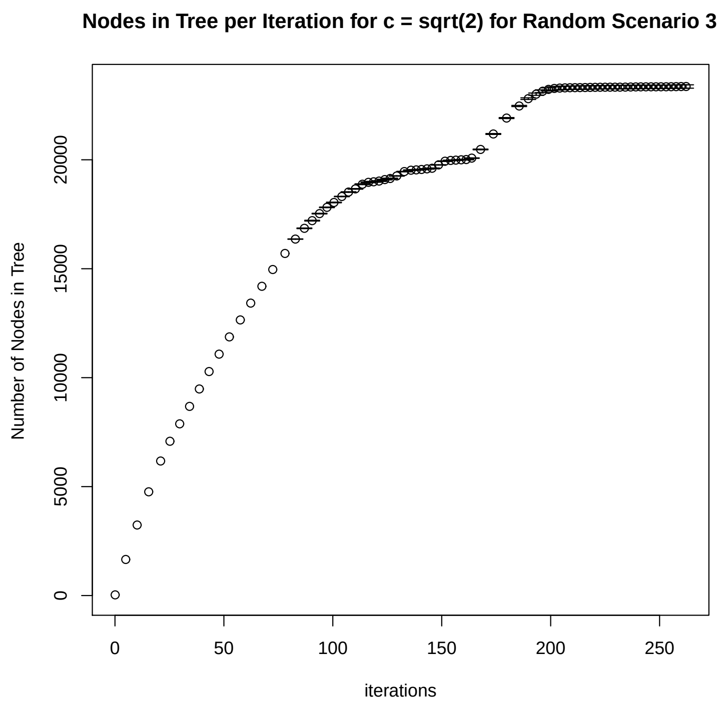


Figure D.7: Average Nodes per time for $c = \sqrt{0.1}$ & $c = \sqrt{1}$ for Random Scenario 3.



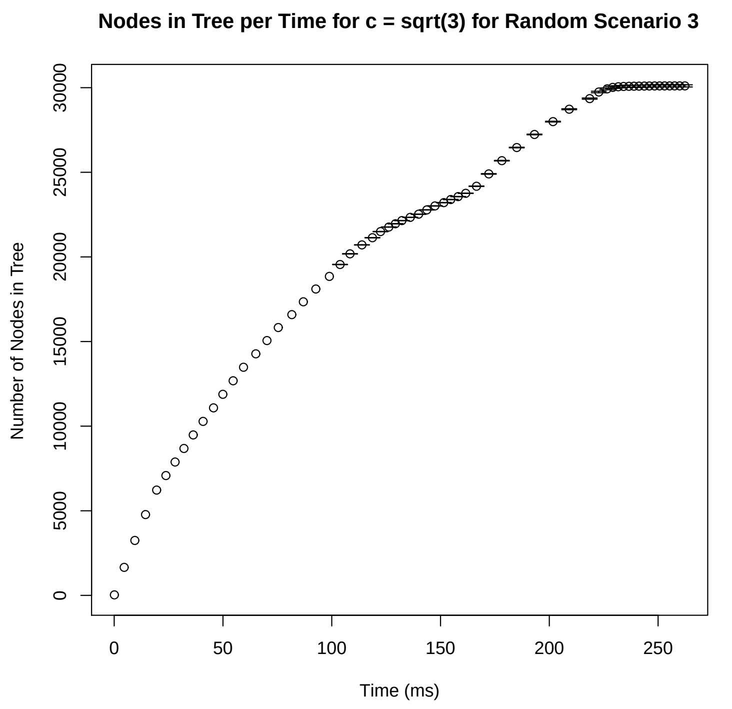
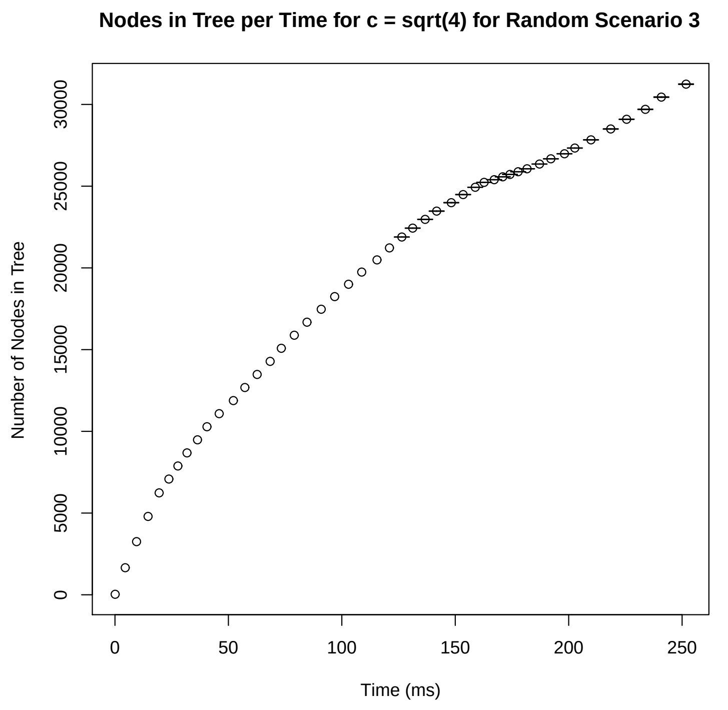


Figure D.8: Average Nodes per time for $c = \sqrt{2}$ & $c = \sqrt{3}$ for Random Scenario 3.



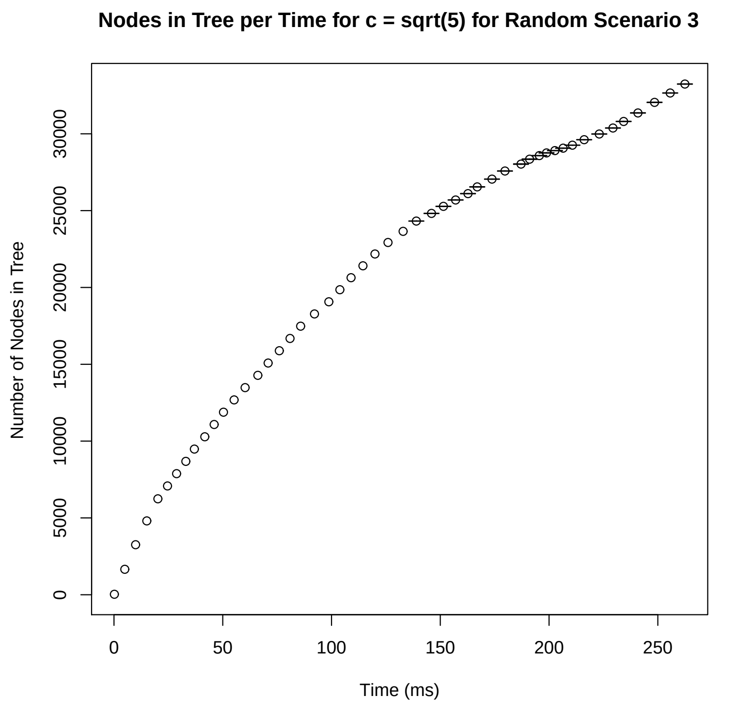


Figure D.9: Average Nodes per time for $c = \sqrt{4}$ & $c = \sqrt{5}$ for Random Scenario 3.

D.4 Random Scenario 4

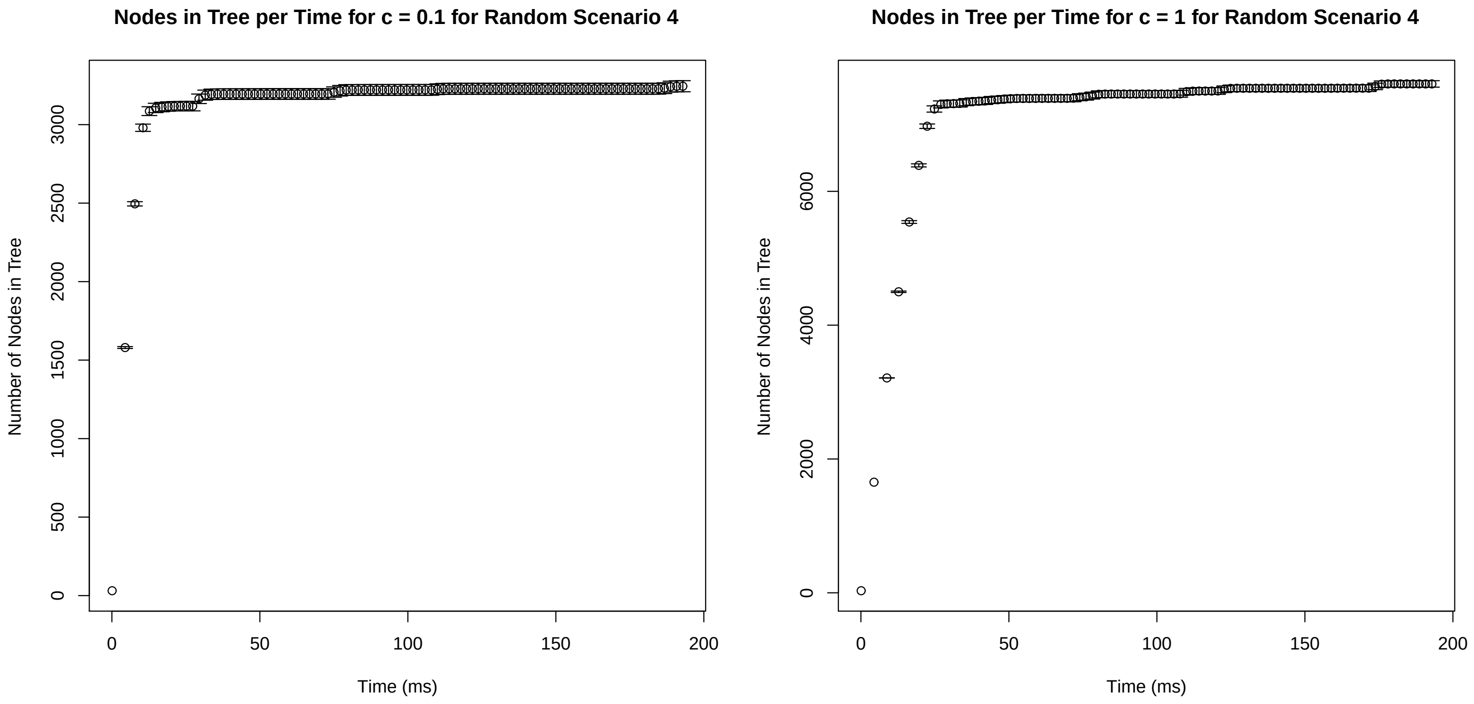
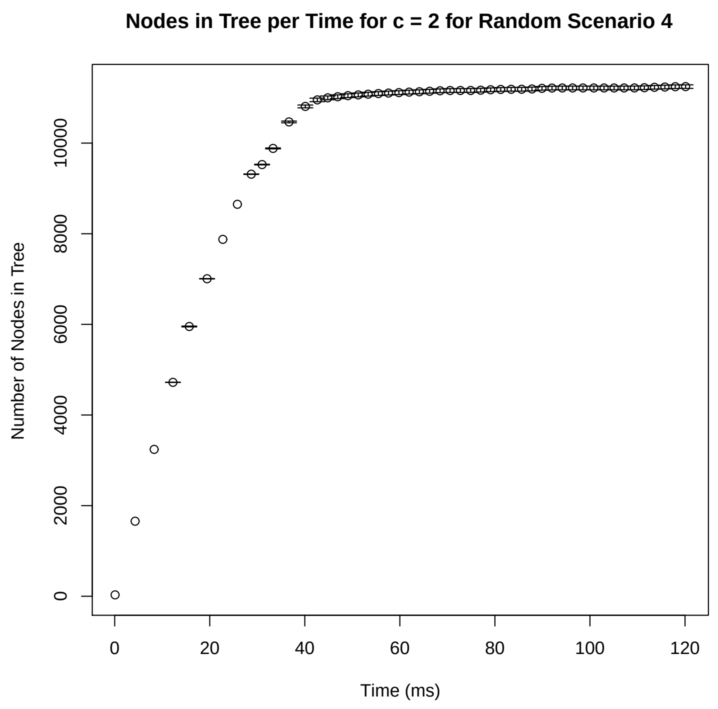


Figure D.10: Average Nodes per time for $c = \sqrt{0.1}$ & $c = \sqrt{1}$ for Random Scenario 4.



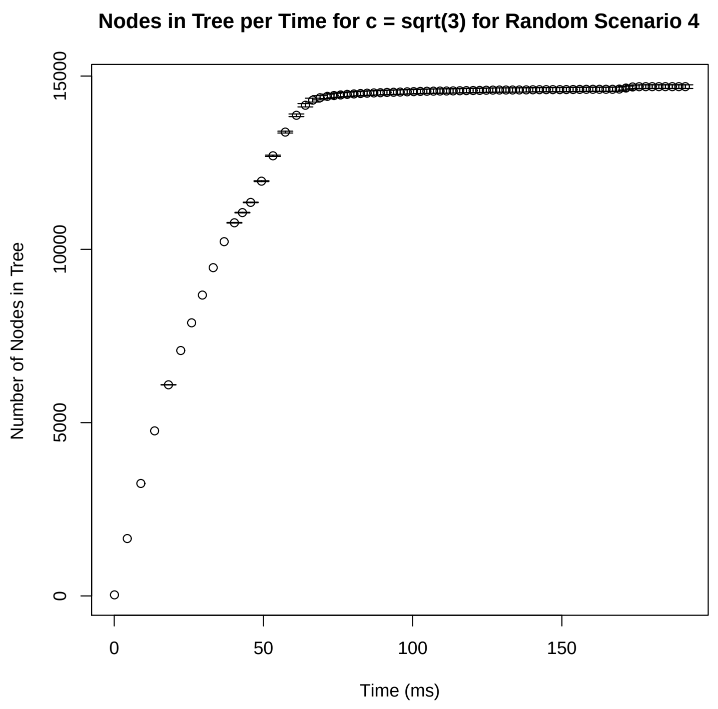
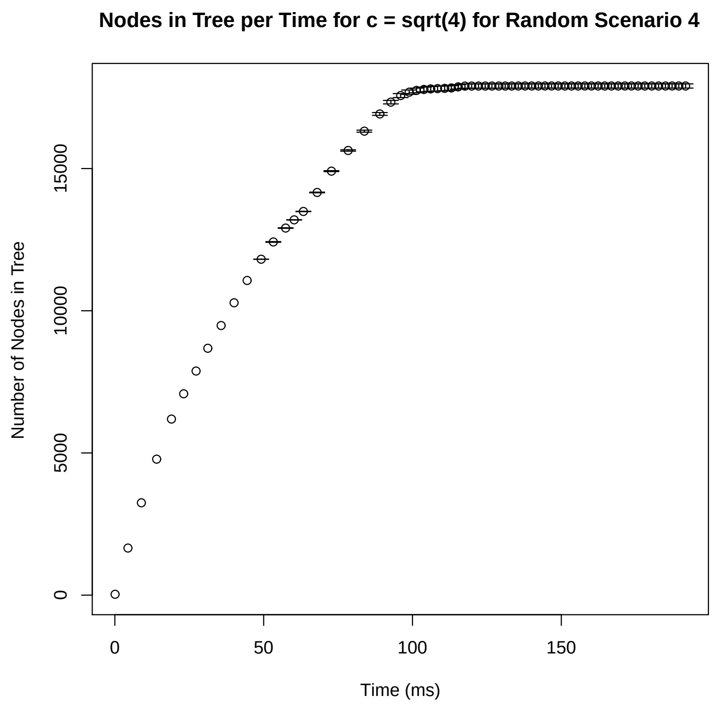


Figure D.11: Average Nodes per time for $c = \sqrt{2}$ & $c = \sqrt{3}$ for Random Scenario 4.



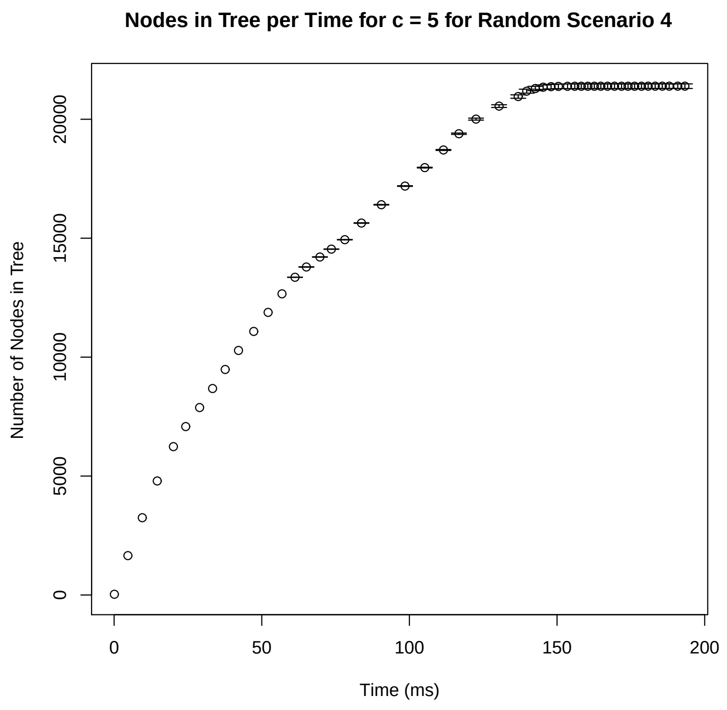
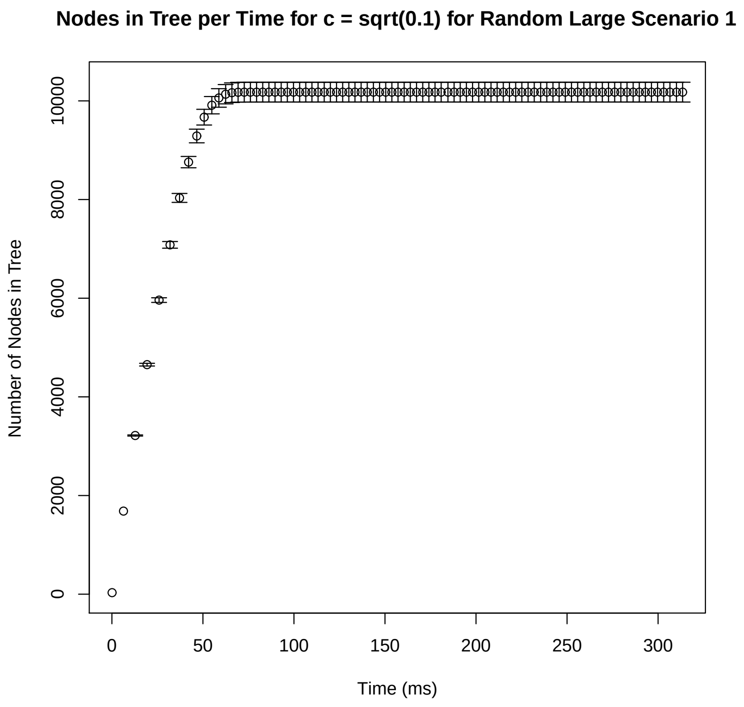


Figure D.12: Average Nodes per time for $c = \sqrt{4}$ & $c = \sqrt{5}$ for Random Scenario 4.

D.5 Random Large Scenario 1



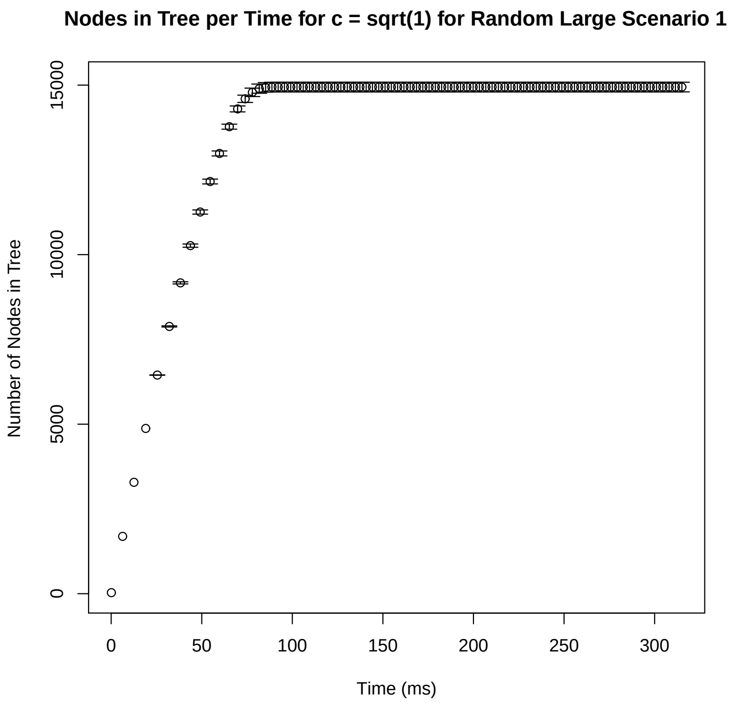


Figure D.13: Average Nodes per time for $c = \sqrt{0.1}$ & $c = \sqrt{1}$ for Random Large Scenario 1.

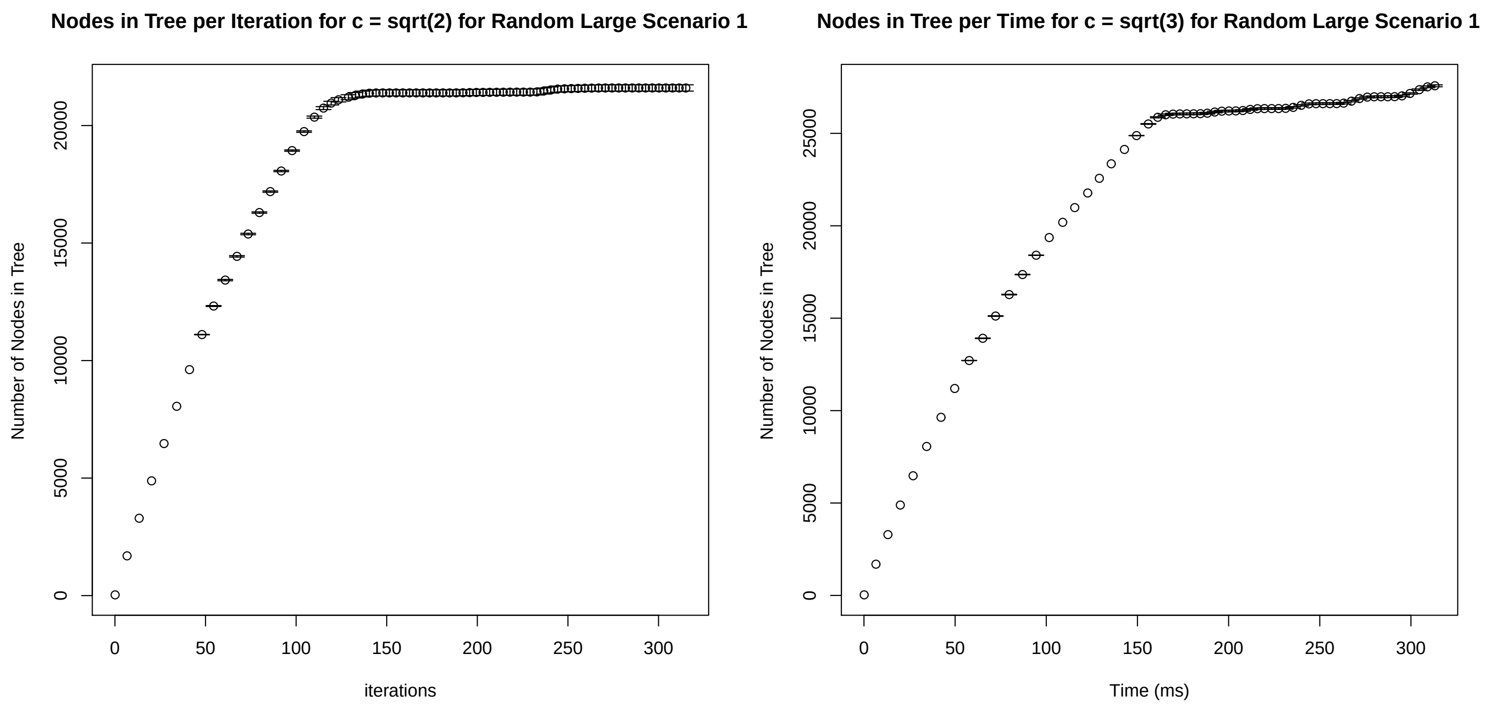


Figure D.14: Average Nodes per time for $c = \sqrt{2}$ & $c = \sqrt{3}$ for Random Large Scenario 1.

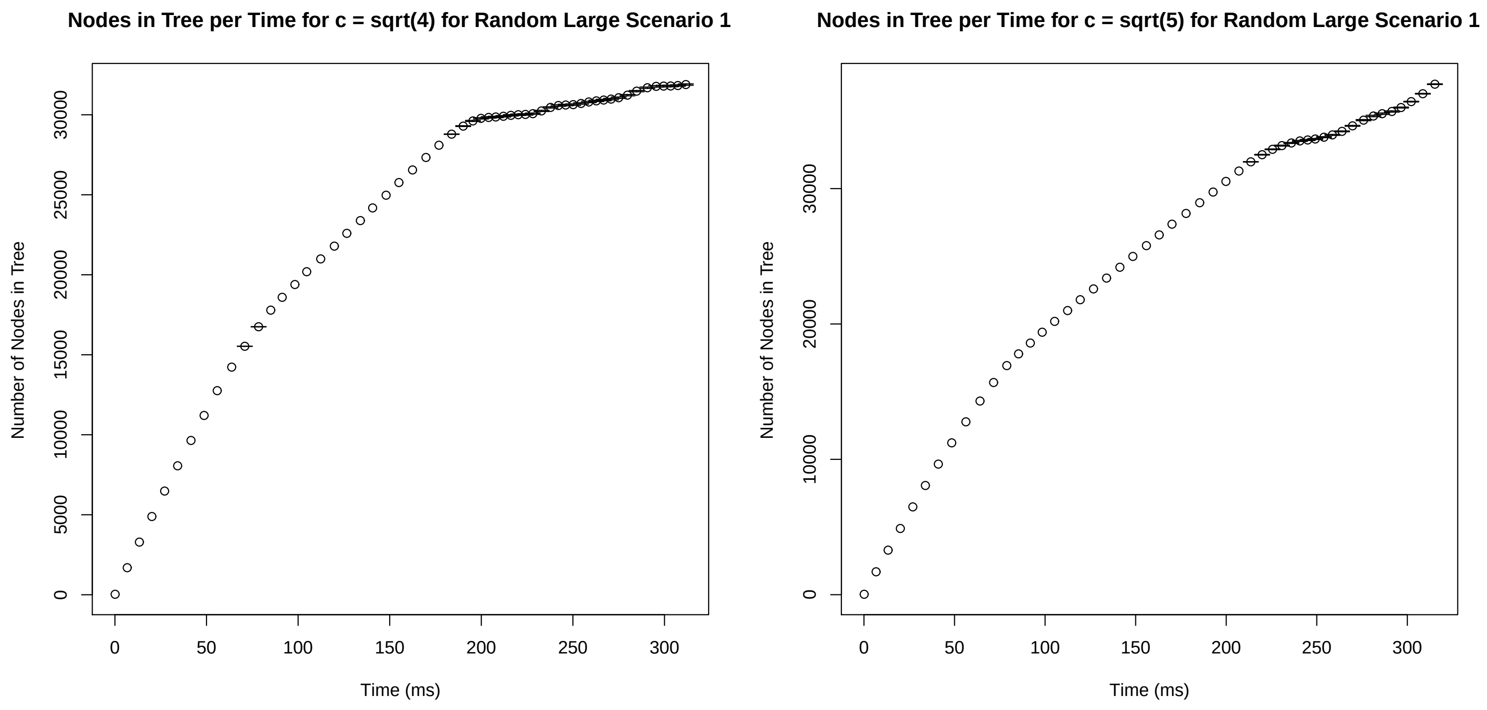
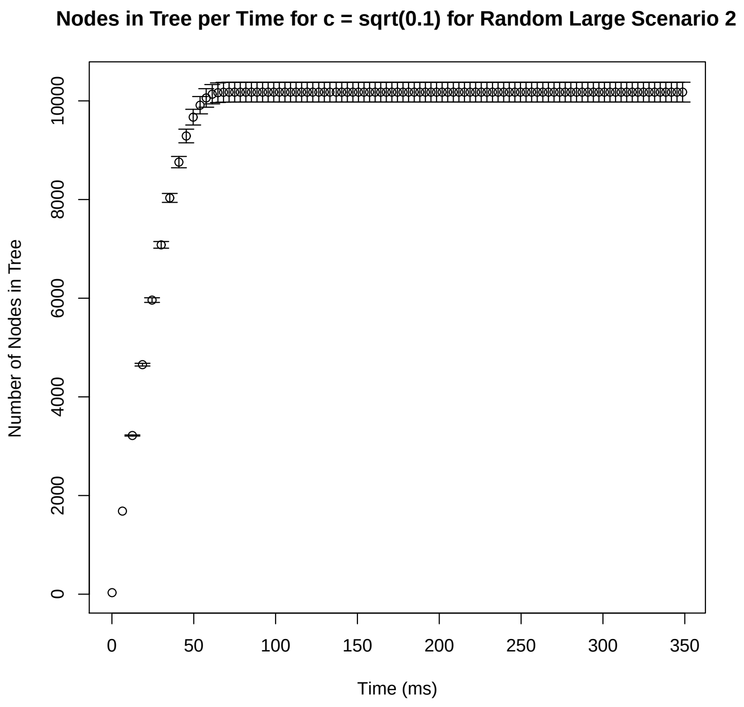


Figure D.15: Average Nodes per time for $c = \sqrt{4}$ & $c = \sqrt{5}$ for Random Large Scenario 1.

D.6 Random Large Scenario 2



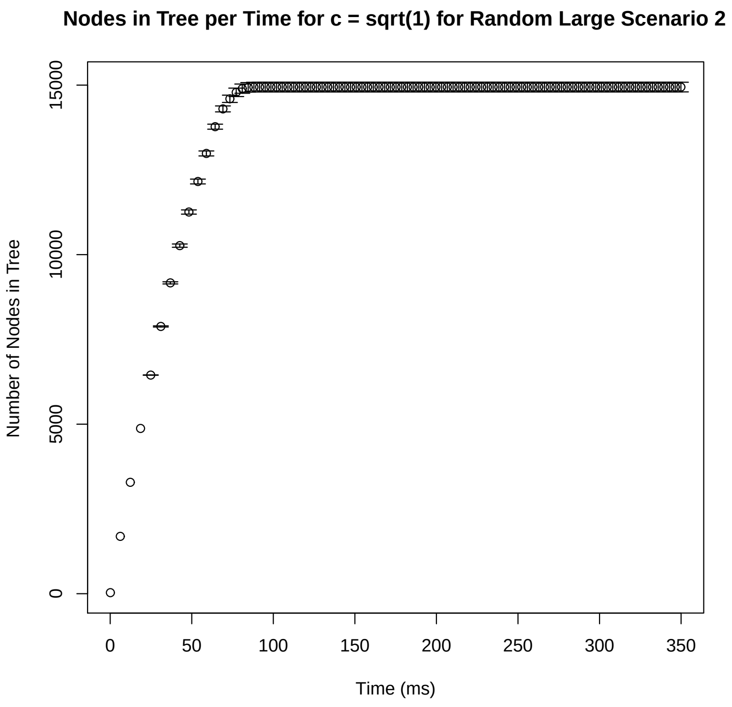
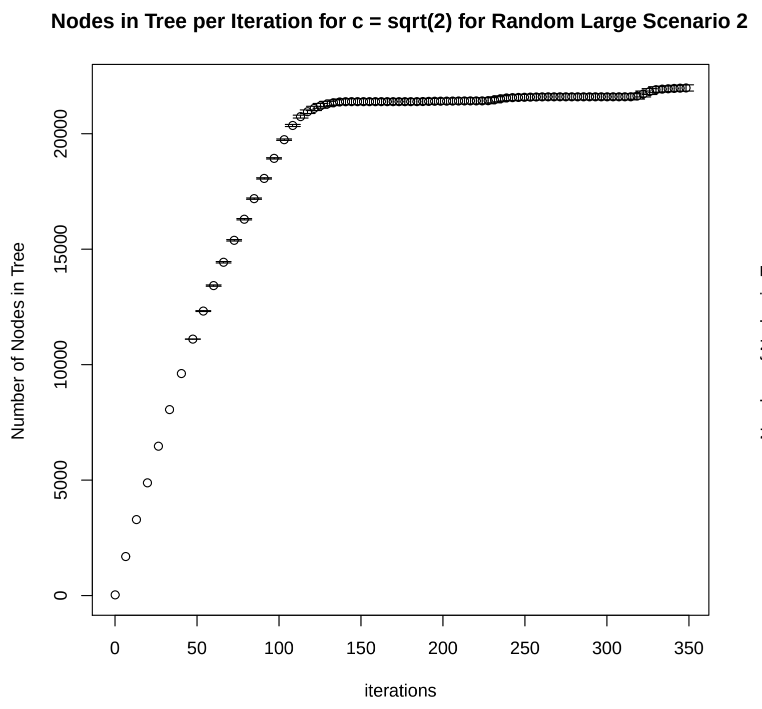


Figure D.16: Average Nodes per time for $c = \sqrt{0.1}$ & $c = \sqrt{1}$ for Random Large Scenario 2.



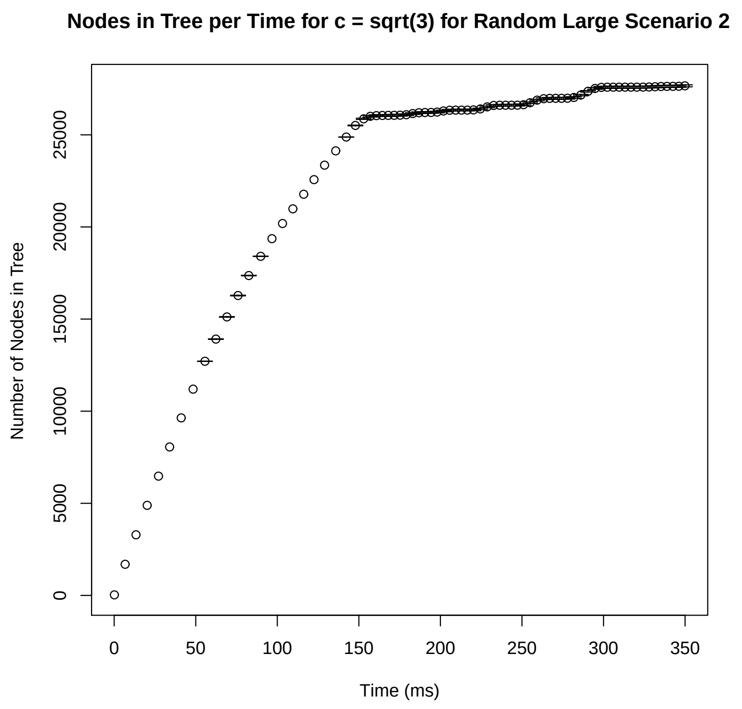
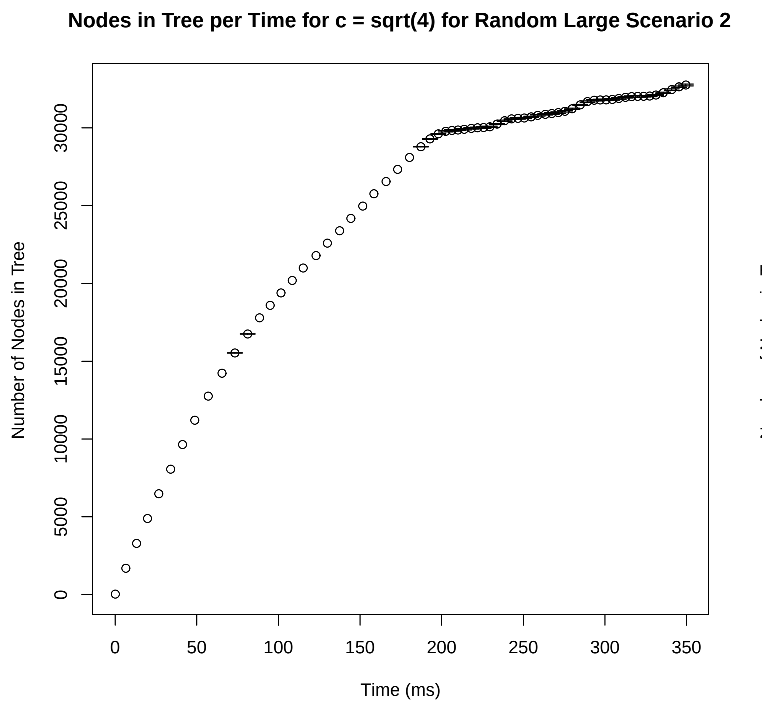


Figure D.17: Average Nodes per time for $c = \sqrt{2}$ & $c = \sqrt{3}$ for Random Large Scenario 2.



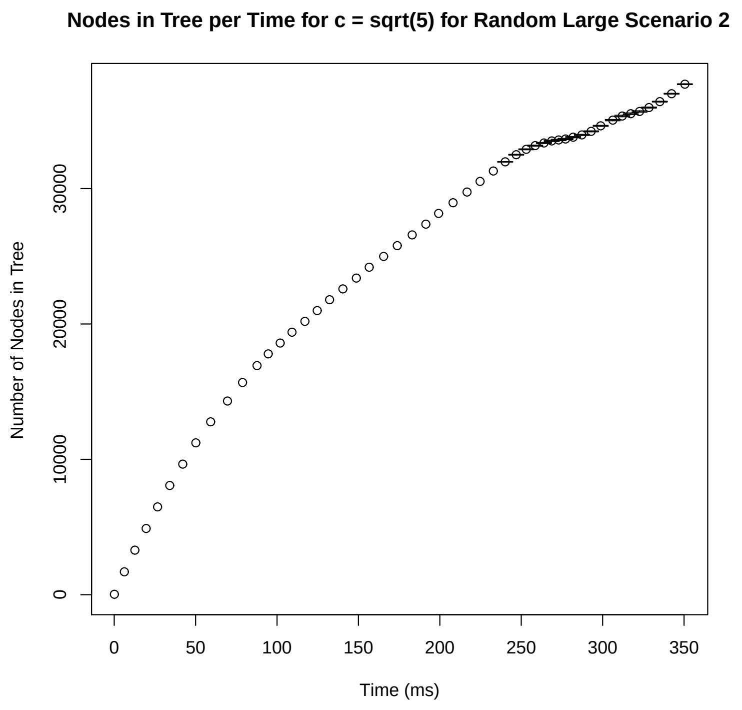
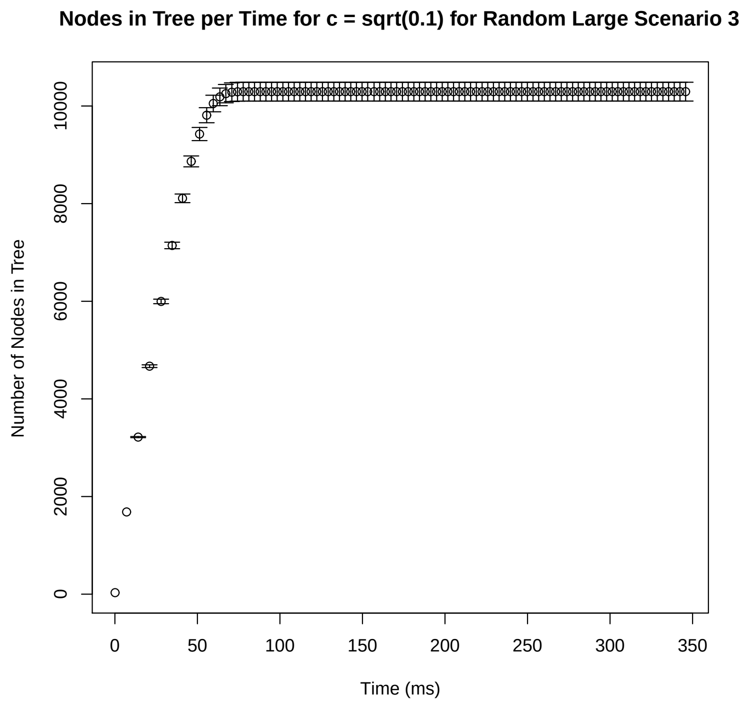


Figure D.18: Average Nodes per time for $c = \sqrt{4}$ & $c = \sqrt{5}$ for Random Large Scenario 2.

D.7 Random Large Scenario 3



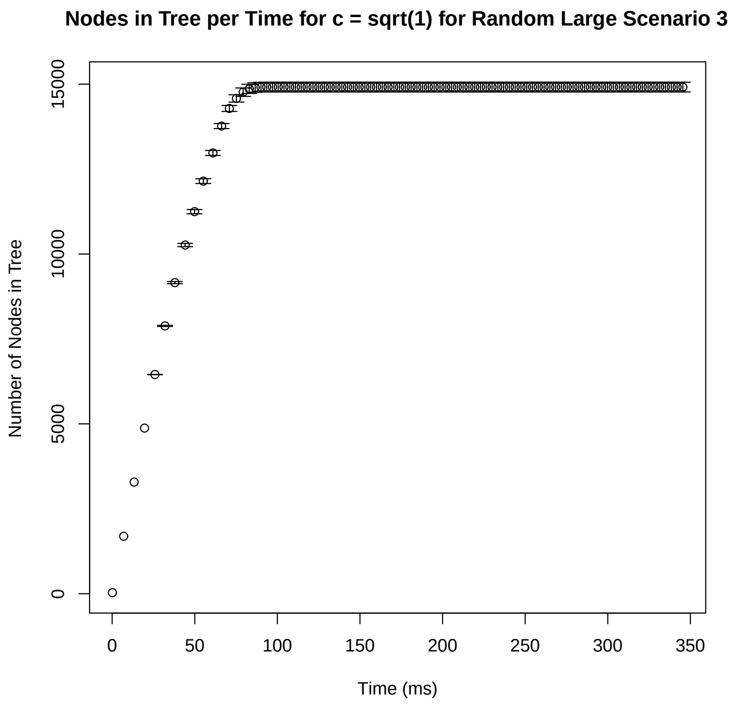
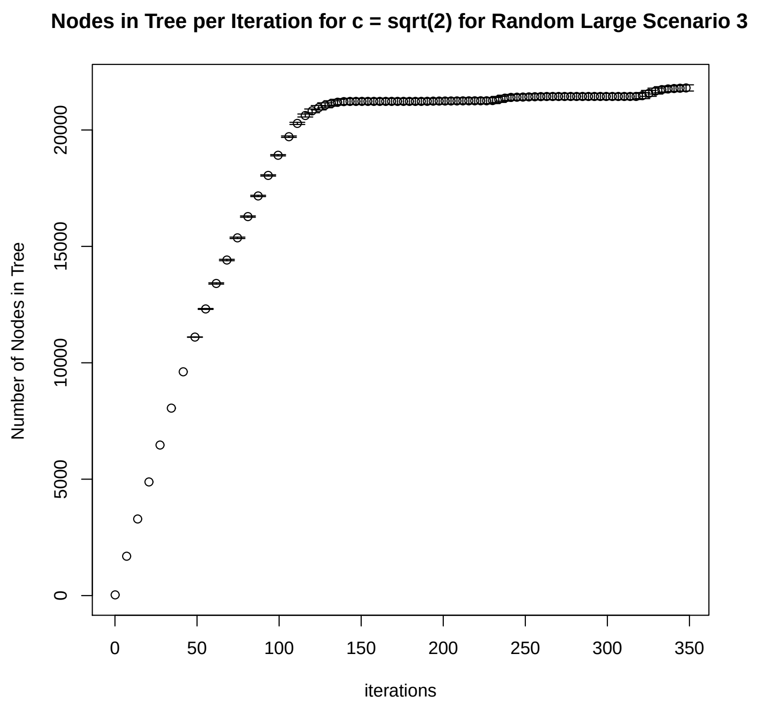


Figure D.19: Average Nodes per time for $c = \sqrt{0.1}$ & $c = \sqrt{1}$ for Random Large Scenario 3.



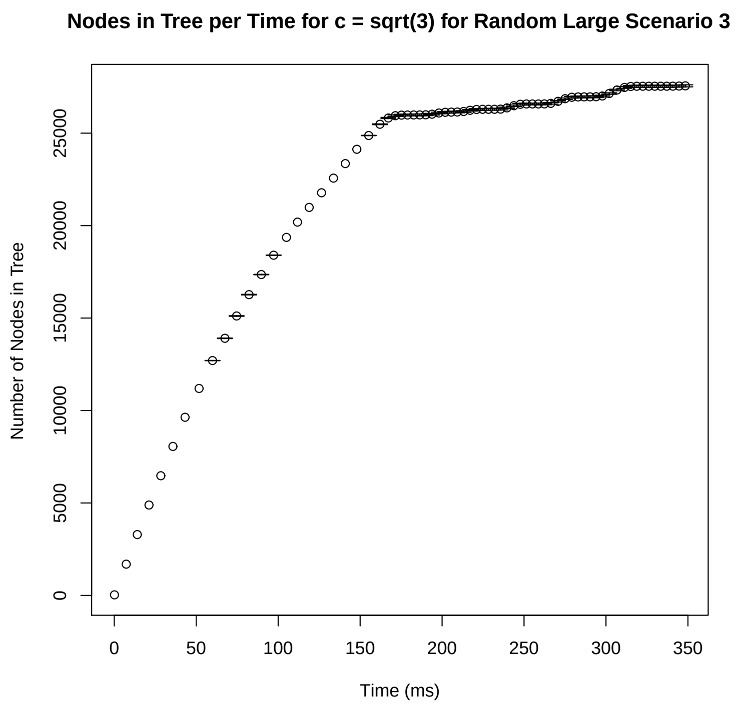
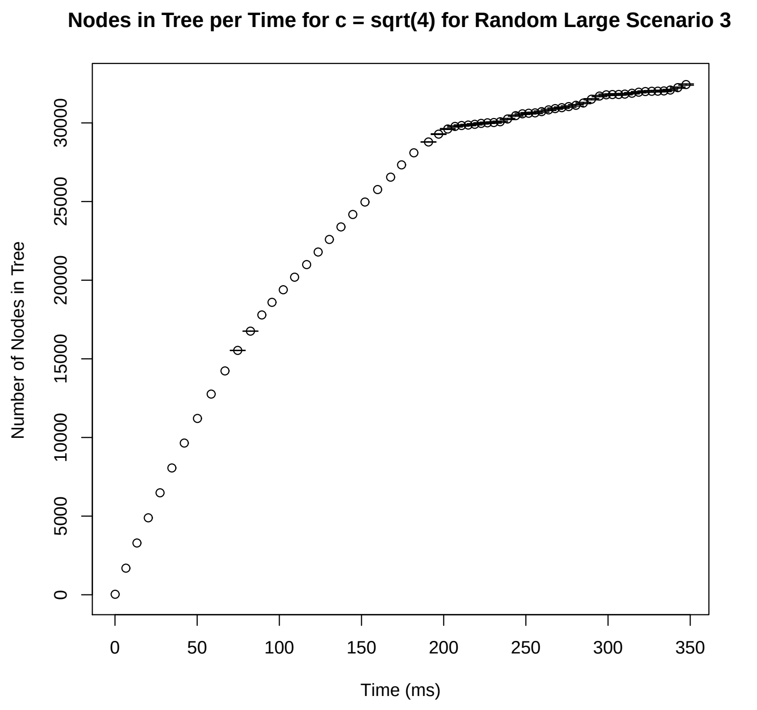


Figure D.20: Average Nodes per time for $c = \sqrt{2}$ & $c = \sqrt{3}$ for Random Large Scenario 3.



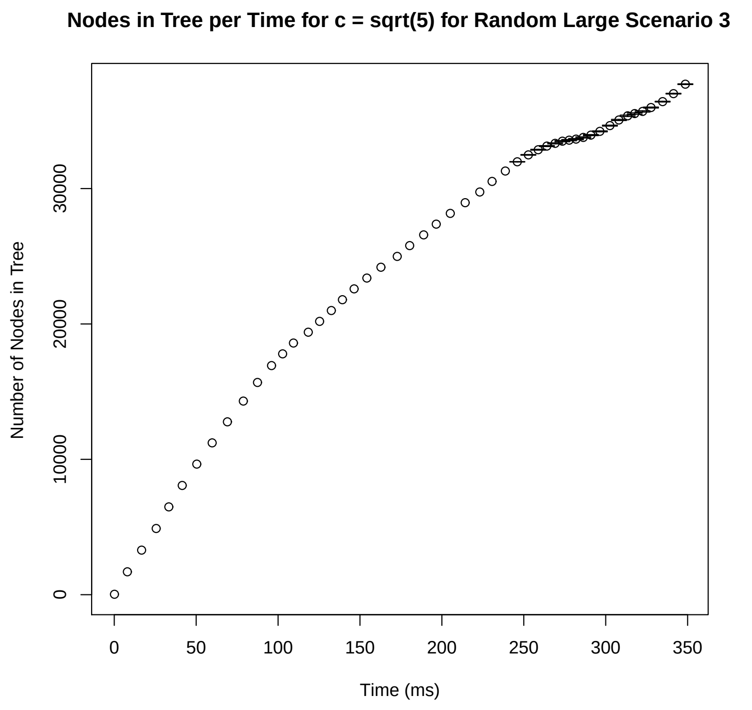
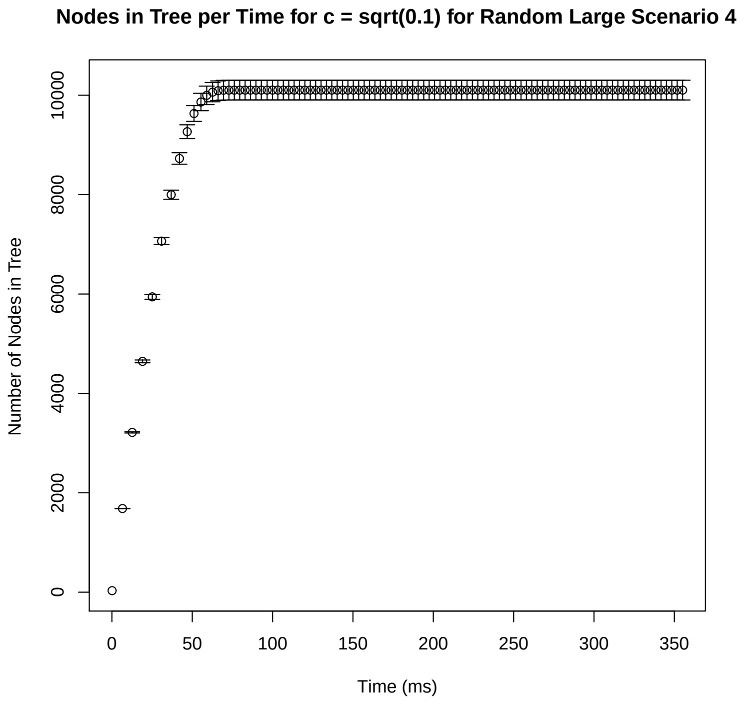


Figure D.21: Average Nodes per time for $c = \sqrt{4}$ & $c = \sqrt{5}$ for Random Large Scenario 3.

D.8 Random Large Scenario 4



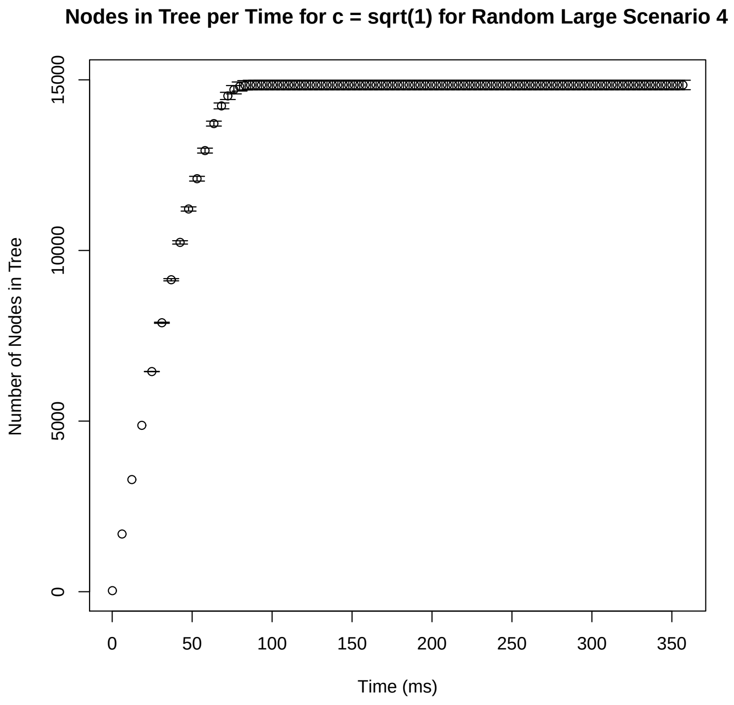
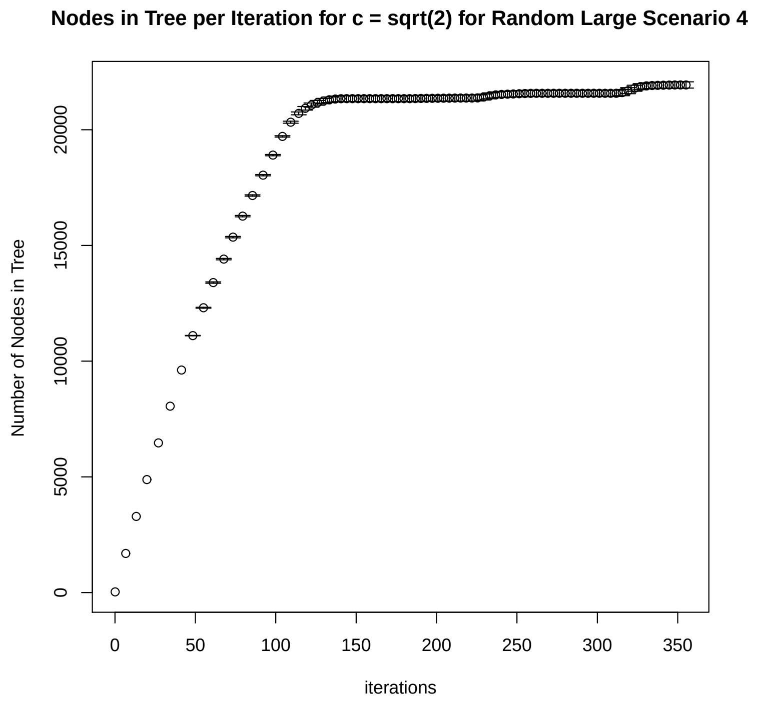


Figure D.22: Average Nodes per time for $c = \sqrt{0.1}$ & $c = \sqrt{1}$ for Random Large Scenario 4.



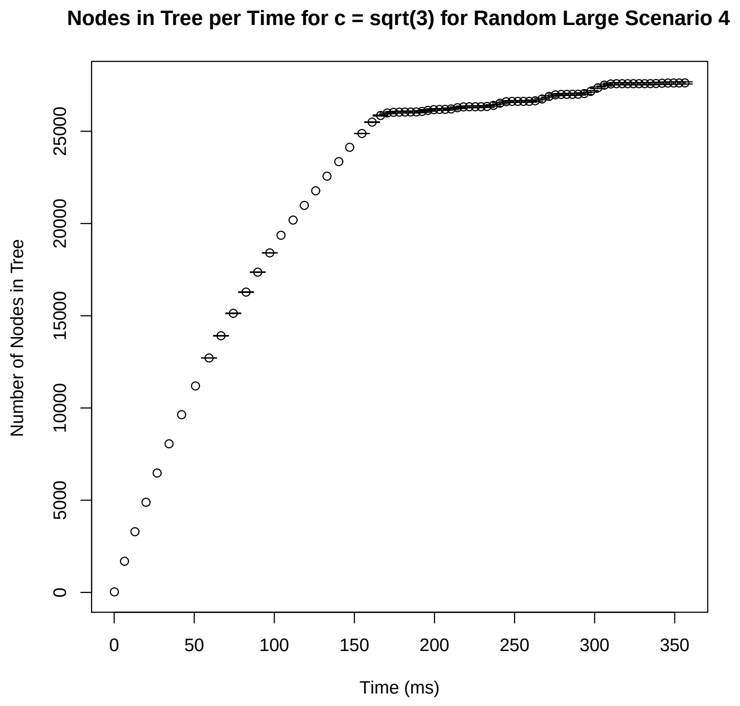
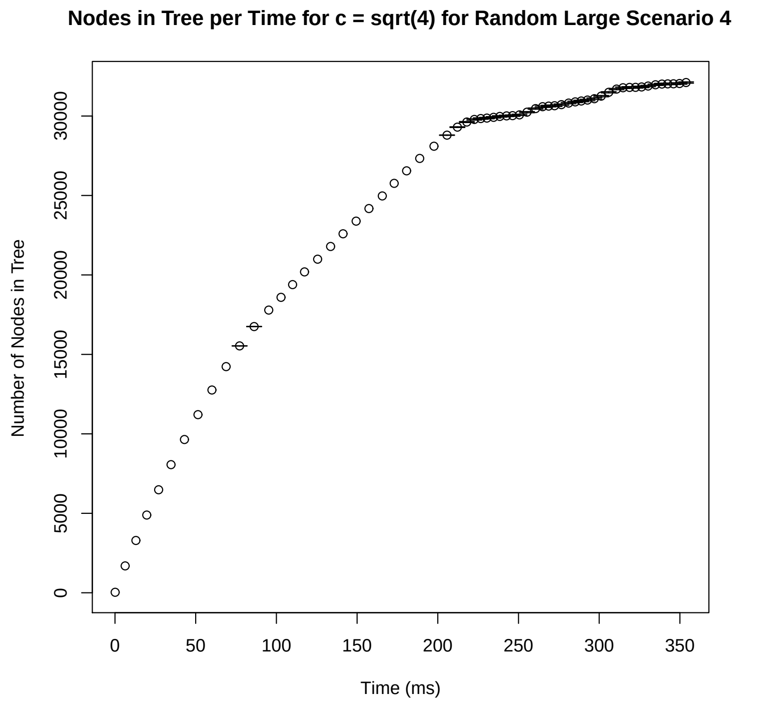


Figure D.23: Average Nodes per time for $c = \sqrt{2}$ & $c = \sqrt{3}$ for Random Large Scenario 4.



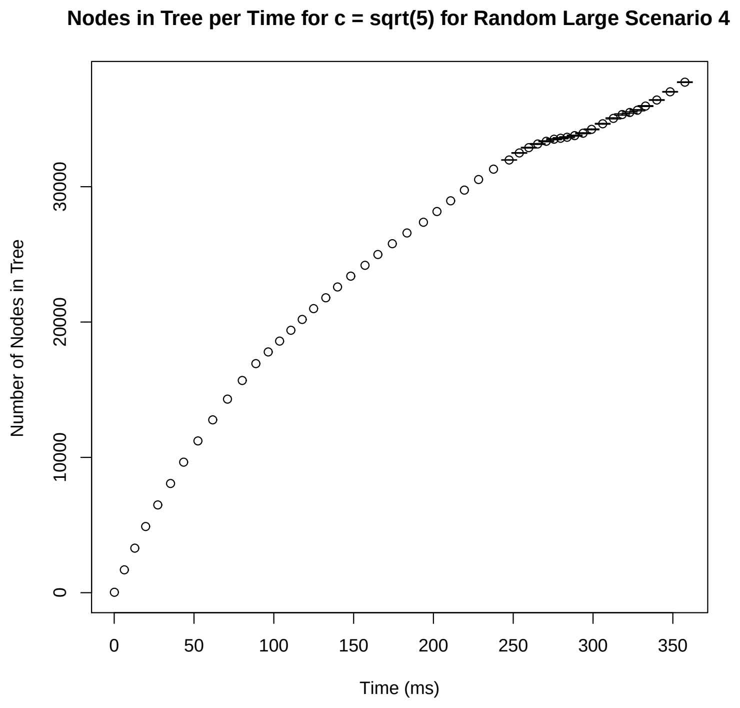


Figure D.24: Average Nodes per time for $c = \sqrt{4}$ & $c = \sqrt{5}$ for Random Large Scenario 4.

Appendix E: Time per Iteration Plots

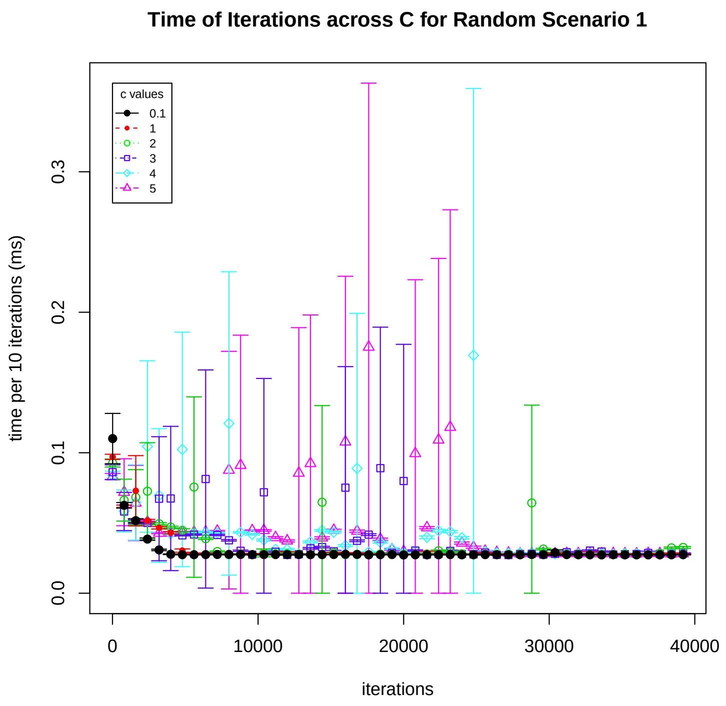


Figure E.1: Average Time for 10 iterations with different *c* values for Random Scenario 1.

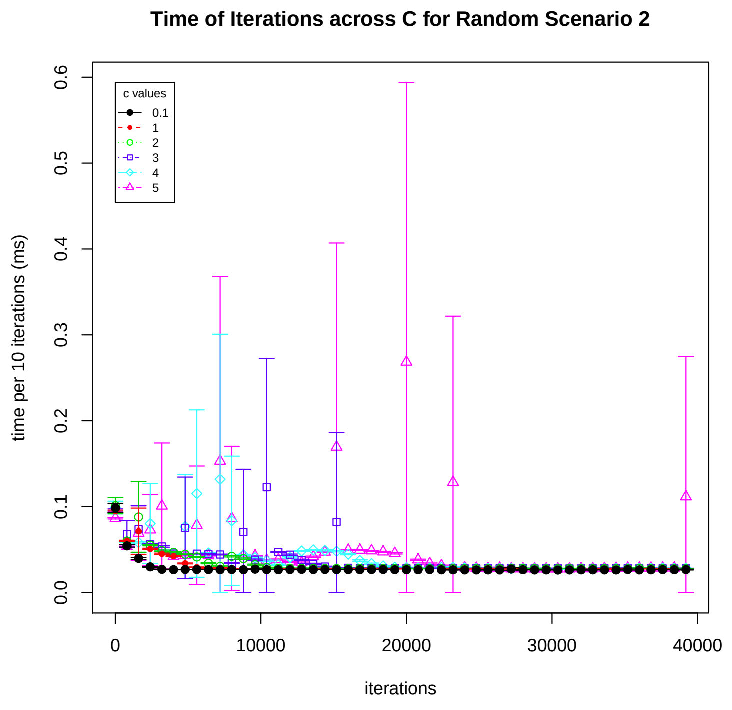


Figure E.2: Average Time for 10 iterations with different *c* values for Random Scenario 2.

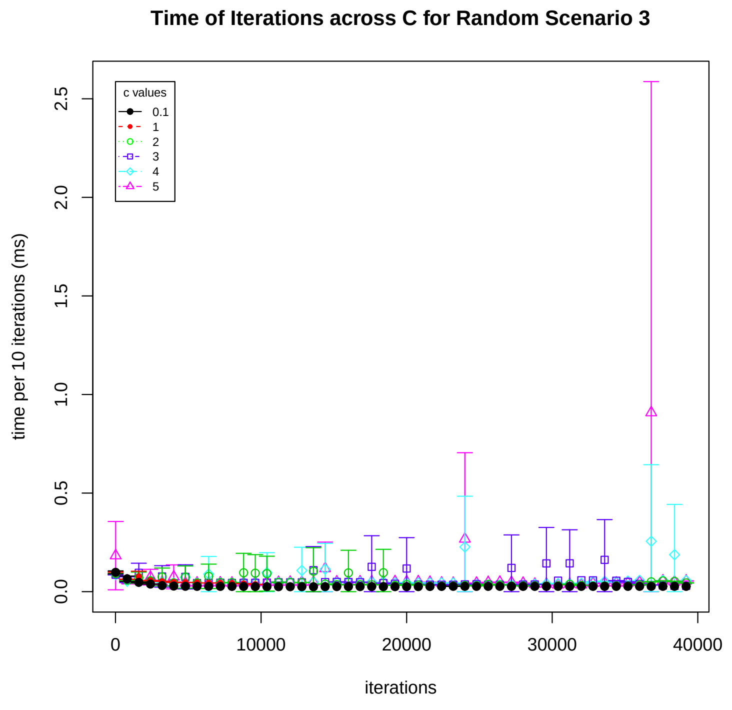


Figure E.3: Average Time for 10 iterations with different *c* values for Random Scenario 3.

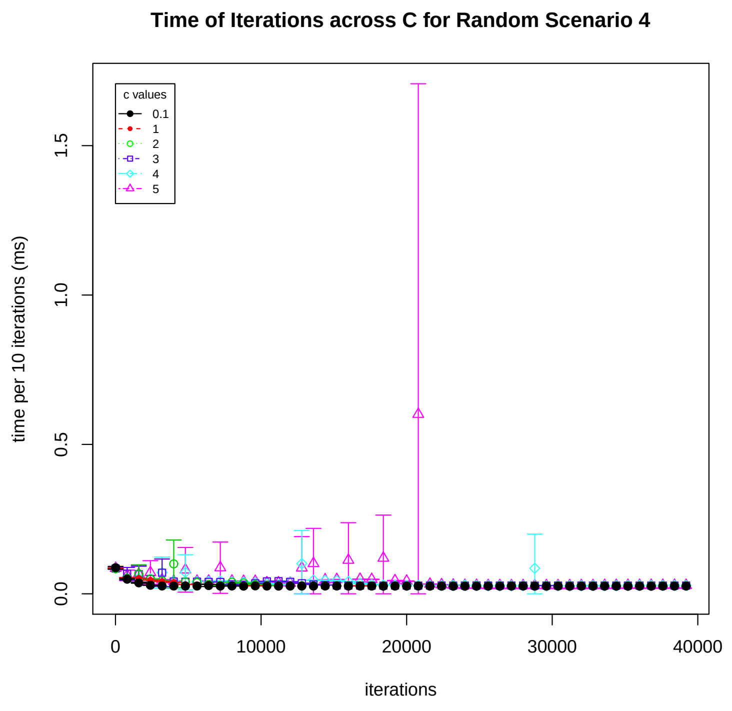


Figure E.4: Average Time for 10 iterations with different *c* values for Random Scenario 4.

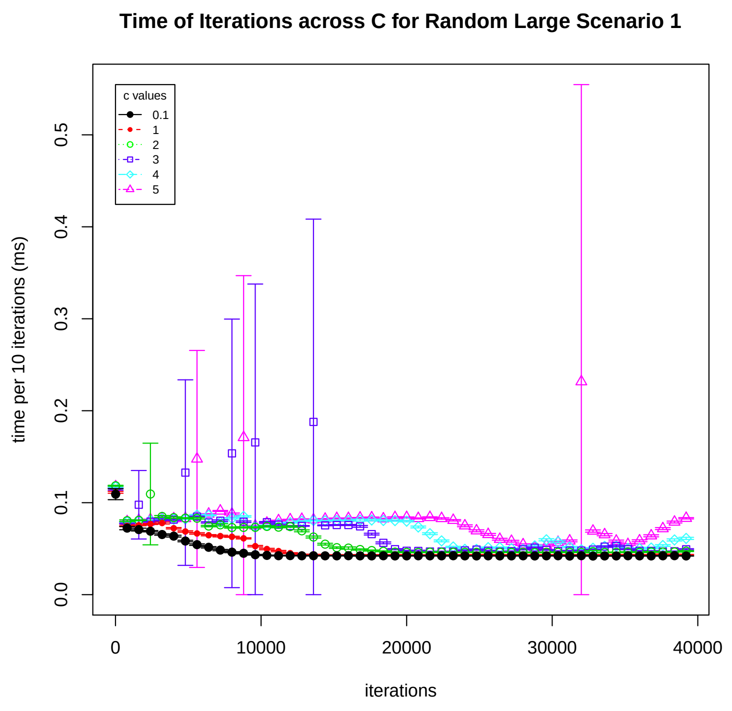


Figure E.5: Average Time for 10 iterations with different *c* values for Large Random Scenario 1.

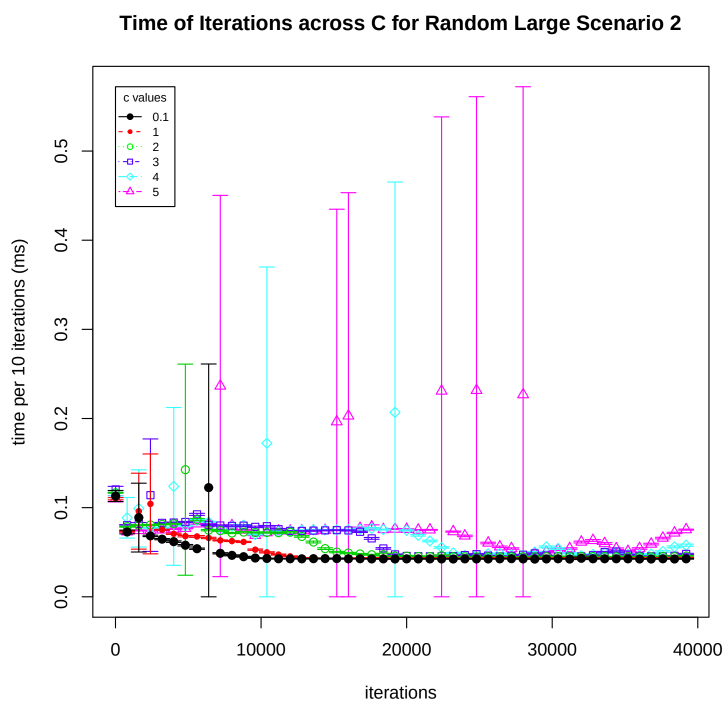


Figure E.6: Average Time for 10 iterations with different c values for Large Random Scenario 2.

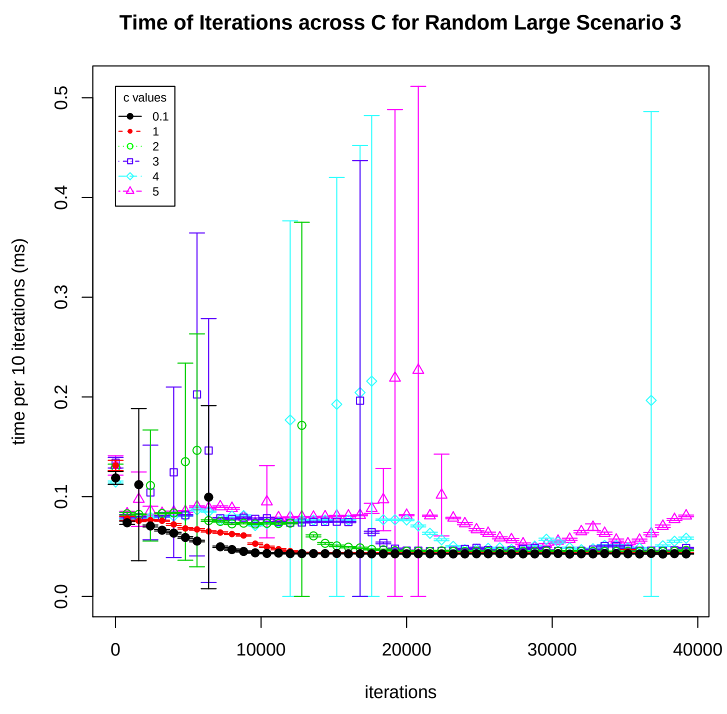


Figure E.7: Average Time for 10 iterations with different c values for Large Random Scenario 3.

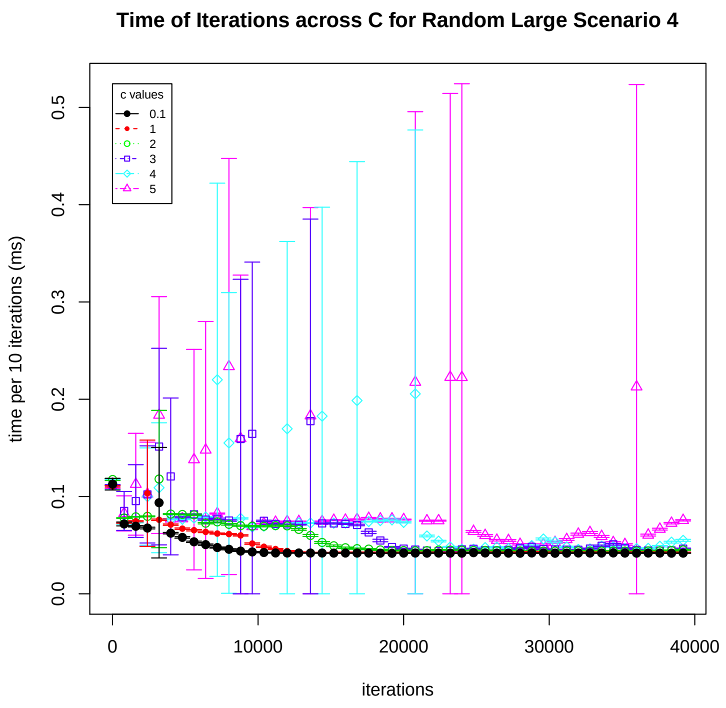


Figure E.8: Average Time for 10 iterations with different c values for Large Random Scenario 4.

Bibliography

[1] Auer, P., N. Cesa-Bianchi, and P. Fischer. "Finite-time analysis of the multiarmed bandit problem". *Machine learning*, 47(2):235–256, 2002.

[2] Balla, R.K. "UCT for tactical assault battles in real-time strategy games". 2009.

[3] Barrett, Anthony, Daniel S. Weld, Oren Etzioni, Steve Hanks, James Hendler, Craig Knoblock, and Rao Kambhampati. "Partial-Order Planning: Evaluating Possible Efficiency Gains". *Artificial Intelligence*, 67:71–112, 1994.

[4] Bellman, R. *A Markovian decision process*. Technical report, DTIC Document, 1957.

[5] Blum, Avrim L. and Merrick L. Furst. "Fast Planning Through Planning Graph Analysis". *Artificial Intelligence*, 90(1):1636–1642, 1995.

[6] Boddy, M., J. Gohde, T. Haigh, and S. Harp. "Course of action generation for cyber security using classical planning". *Proc. of ICAPS*, volume 5. 2005.

[7] Bouguerra, Abdelbaki and Lars Karlsson. "Hierarchical Task Planning under Uncertainty". *In 3rd Italian Workshop on Planning and Scheduling (AI*IA*. 2004.

[8] Bryce, Daniel and Subbarao Kambhampati. "A tutorial on planning graph based reachability heuristics". *AI Magazine*, 2007.

[9] Chen, Kun, Jiuyun Xu, Stephan Reiff-marganiec, and Leicester Le Rh. "Markov-HTN Planning Approach to Enhance Flexibility of Automatic Web Services Composition".

[10] Coates, A., H. Lee, and A.Y. Ng. "An analysis of single-layer networks in unsupervised feature learning". *Ann Arbor*, 1001:48109, 2010.

[11] Cup, KDD. "Available on: http://kdd. ics. uci. edu/databases/kddcup 99/kddcup99. html", 2007.

[12] Dempster, Arthur P, Nan M Laird, and Donald B Rubin. "Maximum likelihood from incomplete data via the EM algorithm". *Journal of the Royal Statistical Society. Series B (Methodological)*, 1–38, 1977.

[13] Erman, Jeffrey, Martin Arlitt, and Anirban Mahanti. "Traffic Classification Using Clustering Algorithms". *Proceedings of the ACM SIGCOMM Workshop on Mining Network Data (MineNet)*. 2006.

[14] Erol, Kutluhan, James Hendler, and Dana S. Nau. "HTN planning: Complexity and expressivity". *In AAAI-94*. 1994.

[15] Erol, Kutluhan, James Hendler, Dana S. Nau, and Reiko Tsuneto. "A critical look at critics in HTN planning". *In Proc. IJCAI-95*. 1995.

[16] Fikes, R.E. and N.J. Nilsson. “STRIPS: A new approach to the application of theorem proving to problem solving”. *Artificial intelligence*, 2(3):189–208, 1972.

[17] Fung, Glenn. “A Comprehensive Overview of Basic Clustering Algorithms”. Technical report, University of Winsconsin, Madison, WI, 2001.

[18] Garner, S.R. et al. “Weka: The waikato environment for knowledge analysis”. *Proceedings of the New Zealand computer science research students conference*, 57–64. Citeseer, 1995.

[19] Garner, Stephen R. “WEKA: The Waikato Environment for Knowledge Analysis”. *In Proc. of the New Zealand Computer Science Research Students Conference*, 57–64. 1995.

[20] Gelly, S., Y. Wang, R. Munos, O. Teytaud, et al. “Modification of UCT with patterns in Monte-Carlo Go”. 2006.

[21] Ghallab, M., C. Aeronautiques, C.K. Isi, D. Wilkins, et al. “PDDL-the planning domain definition language”. 1998.

[22] Ghallab, Malik. *Automated planning : theory and practice*. Elsevier/Morgan Kaufmann, Amsterdam Boston, 2004. ISBN 9781558608566.

[23] Han, Yi, Wen-Xiang Gu, Yang Li, Ming-Hao Yin, and Jing-Bo Zhang. “Flexible Graphplan Based on Heuristic Searching”. *Machine Learning and Cybernetics, 2006 International Conference on*, 160–163. 2006. ID: 1.

[24] Haykin, S.S. *Neural Networks and Learning Machines*. Neural networks and learning machines. Prentice Hall, 3rd edition, 2009. ISBN 9780131471399.

[25] Haykin, S.S., S.S. Haykin, S.S. Haykin, and S.S. Haykin. *Neural networks and learning machines*, volume 3. Prentice Hall, 2009.

[26] Hogg, Chad, Ugur Kuter, and Hctor Muoz-avila. “Learning Hierarchical Task Networks for Nondeterministic Planning Domains”.

[27] Kaelbling, Leslie Pack, Michael L. Littman, and Anthony R. Cassandra. “Planning and acting in partially observable stochastic domains”. *Artificial Intelligence*, 101:99–134, 1998.

[28] Kambhampati, Subbarao. “Design Tradeoffs in Partial Order (Plan Space) Planning”. *In Proc. 2nd Intl. Conf. on AI Planning Systems (AIPS-94*, 92–97. 1994.

[29] Karlik, B. and A.V. Olgac. “Performance analysis of various activation functions in generalized mlp architectures of neural networks”. *International Journal of Artificial Intelligence and Expert Systems*, 1(4):111–122, 2010.

[30] Kira, Kenji and Larry A Rendell. “A practical approach to feature selection”. *Proceedings of the ninth international workshop on Machine learning*, 249–256. Morgan Kaufmann Publishers Inc., 1992.

[31] Kocsis, Levente and Csaba Szepesvri. “Bandit based Monte-Carlo Planning”. *In: ECML-06. Number 4212 in LNCS*, 282–293. Springer, 2006.

[32] Kononenko, Igor, Edvard Simec, and Igor Kononenko Edvard. “Induction of decision trees using ReliefF”, 1995.

[33] Kotsiantis, S. B. “Supervised Machine Learning: A Review of Classification Techniques.” *Informatica*, volume 31, 249–268. 2007.

[34] Laviers, K. and G. Sukthankar. “A real-time opponent modeling system for rush football”. *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Three*, 2476–2481. AAAI Press, 2011.

[35] Lee, P.Y., S.C. Hui, and A.C.M. Fong. “Neural networks for web content filtering”. *Intelligent Systems, IEEE*, 17(5):48–57, 2002.

[36] Lee-urban, Stephen and Hctor Muoz-avila. “Transfer Learning of Hierarchical Task-Network Planning Methods in a Real-Time Strategy Game”.

[37] Li, Yang, Wei-Xiang Gu, Ming-Hao Yin, and Yuan Wasng. “Planning system based on heuristic”. *Machine Learning and Cybernetics, 2005. Proceedings of 2005 International Conference on*, volume 3, 1385–1390 Vol. 3. 2005. ID: 1.

[38] Lin, L.J. *Reinforcement learning for robots using neural networks*. Technical report, DTIC Document, 1993.

[39] MacQueen, J. B. “Some Methods for Classification and Analysis of Multivariate Observations”. *Proceedings of 5th Berkeley Symposium on Mathematical Statisticsand Probability*, 281–297. 1967. URL http://home.dei.polimi.it/matteucc/Clustering/tutorial_html/kmeans.html#macqueen.

[40] McGregor, A., M. Hall, P. Lorier, and J. Brunskill. “Flow clustering using machine learning techniques”. *Passive and Active Network Measurement*, 205–214, 2004.

[41] Menon, A., K. Mehrotra, C.K. Mohan, and S. Ranka. “Characterization of a class of sigmoid functions with applications to neural networks”. *Neural Networks*, 9(5):819–835, 1996.

[42] Nau, Dana. “Current trends in automated planning”. *The AI magazine*, 28(4):43–58, 2007. Doi: pmid:.

[43] Nau, Dana, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. “SHOP2: An HTN planning system”. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

[44] Nau, Dana S., Stephen J. J. Smith, and Kutluhan Erol. "Control Strategies in HTN Planning: Theory Versus Practice". *In AAAI-98/IAAI-98 Proceedings*, 1127–1133. AAAI Press, 1998.

[45] Obama, B. "Comprehensive national cybersecurity initiative", 2010.

[46] Pellier, D., B. Bouzy, and M. Metivier. "Using a Goal-Agenda and Committed Actions in Real-Time Planning". *Tools with Artificial Intelligence (ICTAI), 2011 23rd IEEE International Conference on*, 74–81. 2011. ISBN 1082-3409. ID: 1.

[47] Quinlan, J. R. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers, 1993. ISBN 9781558602380.

[48] Russell, Stuart. *Artificial intelligence : a modern approach*. Prentice Hall, Upper Saddle River, 2010. ISBN 9780136042594.

[49] Saarinen, S., R. Bramley, and G. Cybenko. "Neural networks, backpropagation, and automatic differentiation". *Automatic Differentiation of Algorithms: Theory, Implementation and Application, SIAM Proceedings Series List*, 31–42, 1991.

[50] Shun, J. and H.A. Malki. "Network intrusion detection system using neural networks". *Natural Computation, 2008. ICNC'08. Fourth International Conference on*, volume 5, 242–246. IEEE, 2008.

[51] Sohrabi, Shirin, Jorge A. Baier, and Sheila A. Mcilraith. "HTN Planning with Preferences".

[52] Tavallaee, M., E. Bagheri, Wei Lu, and A. A. Ghorbani. "A detailed analysis of the KDD CUP 99 data set". *Computational Intelligence for Security and Defense Applications, 2009. CISDA 2009. IEEE Symposium on*, 1–6. 2009. ID: 1.

[53] Weld, Daniel S. "An introduction to least commitment planning". *AI Magazine*, 1994.

[54] Werbos, P. "Beyond regression: New tools for prediction and analysis in the behavioral sciences". 1974.

[55] Widrow, B., M.E. Hoff, et al. "Adaptive switching circuits." 1960.

[56] Zander, Sebastian, Thuy Nguyen, and Grenville Armitage. "Automated Traffic Classification and Application Identification using Machine Learning". *Proceedings of the The IEEE Conference on Local Computer Networks 30th Anniversary*, LCN '05, 250–257. IEEE Computer Society, Washington, DC, USA, 2005. ISBN 0-7695-2421-4. URL http://dx.doi.org/10.1109/LCN.2005.35.

[57] Zander, Sebastian, Thuy Nguyen, and Grenville Armitage. "Self-learning IP traffic classification based on statistical flow characteristics". *Passive and Active Network Measurement*, 325–328, 2005.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704–0188*

1. REPORT DATE (DD–MM–YYYY) 27-03-2013	2. REPORT TYPE Master's Thesis	3. DATES COVERED (*From — To*) Oct 2011 – Mar 2013

4. TITLE AND SUBTITLE Development of a Response Planner using the UCT Algorithm for Cyber Defense	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Knight, Michael P., Captain, USAF	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/ENY) 2950 Hobson Way WPAFB OH 45433-7765	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT-ENG-13-M-28
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Dr. Richard Fedors AETC AFRL/RISF 525 Brooks Road Rome, NY 13441-4505 (315) 330-3608 Richard.fedors@rl.af.mil	10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RISF
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

A need for a quick response to cyber attacks is a prevalent problem for computer network operators today. There is a small window to respond to a cyber attack when it occurs to prevent significant damage to a computer network. Automated response planners offer one solution to resolve this issue. This work presents Network Defense Planner System (NDPS), a planner dependent on the effectiveness of the detection of the cyber attack. This research first explores making classification of network attacks faster for real-time detection, the basic function Intrusion Detection System (IDS) provides. After identifying the type of attack, learning the rewards to use in the NDPS is the second important area of this research.

For NDPS to assemble the optimal plan, learning the rewards for resulting network states is critical and often depends on the preferences of the network operator. Using neural networks, the second area of this research demonstrates that capturing the preferences through samples is feasible. After training the neural network, a model can be created to obtain reward estimates. The research performed in these two areas complement the final portion of the research which is assembling the optimal plan through using the Upper Bounds on Confidence for Trees (UCT) algorithm.

NDPS is implemented using the UCT algorithm which allows for quick plan formulation by searching through predicted network states based on available network actions. UCT can effectively create a plan quickly and is guaranteed to provide the optimal plan, according to rewards used, if enough time is allotted. NDPS is tested against eight random attack scenarios. For each attack scenario, the plan is polled at specific time intervals to test how quickly the optimal plan can be formulated. Results demonstrate the feasibility of NDPS to be used in real world scenarios since the optimal plans for each attack type can be formulated in real-time allowing for a rapid system response.

15. SUBJECT TERMS
UCT, automated planning, classifier, cyber defense, neural network

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON Major Kennard Laviers
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U	UU	153	19b. TELEPHONE NUMBER (Include Area Code) (937)255-3636, ext 4395

Standard Form 298 (Rev. 8–98)
Prescribed by ANSI Std. Z39.18